Report No. 367

ISL-A SEMANTICS LANGUAGE FOR A TRANSLATOR WRITING SYSTEM

by

Nelson Castro Machado

December 11, 1969

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 367

ISL-A SEMANTICS LANGUAGE FOR A TRANSLATOR WRITING SYSTEM

by

Nelson Castro Machado

December 11, 1969

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

* This work was supported in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. USAF 30(602)-4144 and submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, August 1969.

ABSTRACT

It is well known that a Translator Writing System (TWS) can, in general, be divided into two parts: a syntax preprocessor and a semantics preprocessor, yielding compilers composed of a syntactic recognizer and a package of semantic routines. This paper describes the semantic preprocessor of a TWS. Semantics is specified as a program written in a special purpose language called ISL which is basically an extension of ALGØL. The extensions include some constructs often needed in the description of semantics of a language, such as stack and table declaration and manipulation, and control words to automate to a great extent the use of the TWS. The structure and use of ISL are described and an example of the use of ISL to specify the semantics of a small subset of ALGØL is presented and discussed.

ACKNOWLEDGMENTS

I wish to express my deep gratitude to Professor Robert S. Northcote for the aid and encouragement in both the structure of the language herein described and the content of this document.

Professor David J. Kuck is also to be thanked for his guidance in the initial phase of this work, as are my colleagues Alan J. Beals, Jacques LaFrance and Robert Mercer for the many helpful suggestions.

I would like to express my appreciation for the financial support given by the Department of Computer Science and the ILLIAC IV Project.

Finally, special thanks are due to Mrs. Patricia Douglas and Mrs. Kay Flessner for the efficient typing of the final document, and to my wife Arlene for patience and incentive.

TABLE OF CONTENTS

Page

1. INTRODUCTION 1

2. THE STRUCTURE OF THE TWS 8

2.1 Basic Characteristics 8

2.1.1 Basic Block Diagram 9

2.1.2 The Intermediate Language 9

2.2 The Structure of Translators Generated by the System 13

2.2.1 One-pass Compilers 13

2.2.1.1 The Scanner and BIGTAB 15

2.2.1.2 The Parser and MSTACK 26

2.2.1.3 The Semantic Routines 28

2.2.2 Multi-pass Compilers - the IL Recognizer 28

2.3 The Actual System 31

2.3.1 Detailed Block Diagram 31

2.3.2 TWINKLE and the Syntax-Semantics Links 34

2.3.2.1 Semantic Action Calls 37

2.3.2.2 Semantic Tests 39

2.3.2.3 Other Links 40

2.3.3 The Floyd Productions Generator 42

2.3.4 The Parser Tables Generator 42

2.3.5 Interpretive and Executable Parts - The ALGOL Code Generator 42

2.3.6 The ISL Translator 44

3. DESIGN AND IMPLEMENTATION 46

3.1 Design Considerations 46

3.1.1 Critique of FSL 46

3.1.2 The ISL Approach 47

Page

3.2 Implementation 49

3.2.1 The Brute-Force ISL Translator 49

3.2.1.1 The Basic Parsing Algorithm 50

3.2.1.2 The Parsing Algorithm as Applied to the Implementation of a Class of ALGOL Extensions 52

3.2.2 The TWS-ISL Translator 55

4. A DESCRIPTION OF ISL 56

4.1 General Structure of ISL 56

4.1.1 The Action Label 57

4.1.2 Semantic Action Zero 59

4.1.3 The Structure of a Translated ISL Program 59

4.2 ISL Declarations 62

4.2.1 Data Structure Declarations 62

4.2.1.1 Table Declarations 62

4.2.1.2 Stack Declarations 65

4.2.1.3 Cell Declarations 67

4.2.1.4 The Predefined Data Structures and the Add Column Declaration 68

4.2.2 Control Declarations 69

4.2.2.1 Forward Action Declaration 69

4.2.2.2 Number of Actions Declaration 71

4.2.2.3 In Declaration 72

4.2.2.4 End of Declarations Declaration 73

4.2.3 Define Declarations and Definition Calls 73

4.3 ISL Operands 75

4.3.1 The ISL Operand 75

4.3.2 Pointer Operand 77

4.3.3 Search Operand 78

Page

4.4 ISL Statements 78

4.4.1 Increment and Decrement Statements 79

4.4.2 Execute Statement 79

4.4.3 Push and Pop Statements 80

4.4.4 Enter Statement 81

4.4.5 Code Statement 82

5. USE OF THE ISL TRANSLATOR 86

5.1 The File Structure 86

5.1.1 Block Diagram 86

5.1.2 Selection of the Input File 90

5.2 Running the Translator 91

5.2.1 Indirect Run 91

5.2.2 Direct Run 93

5.3 ISL Control Cards and Control Options 93

5.3.1 The Language Options 96

5.3.2 The Listing Options 96

5.3.3 The Sequencing Options 97

5.3.4 The Multi-pass Options 99

5.3.5 The Zipping Options 100

5.3.6 The Option SPSTAB 105

6. USE OF ISL - AN EXAMPLE 106

6.1 General Considerations 106

6.2 Analysis of the Semantic Description of DEMALGOL I 108

6.2.1 Dividing the Semantics into Passes 108

6.2.2 Planning the Structure of a Pass 112

6.2.3 Assigning Semantic Calls to the Syntax and Writing the Semantic Actions 114

Page

APPENDIX

A. FORMAL SYNTAX OF ISL 119
B. SYNTAX OF DEMALGOL I IN TWINKLE - BNF FORM 129
C. SYNTAX OF DEMALGOL I IN TWINKLE - ENGLISH-LIKE FORM 132
D. SEMANTICS OF DEMALGOL I IN ISL - PASS 1 135
E. OUTPUT PRODUCED BY THE ISL TRANSLATOR ON THE SEMANTICS OF DEMALGOL I - PASS 1. 142

LIST OF REFERENCES 150

LIST OF FIGURES

Figure		Page
1.	Classical structure of a TWS	3
2.	General structure of the ILLIAC IV TWS	10
3.	Block diagram of the IL recognizer	11
4.	ALGOL block structure of a one-pass compiler	14
5.	Format of words in BIGTAB	18
6.	Sample entry of an identifier in BIGTAB	19
7.	Format of the words in MSTACK	27
8.	ALGOL block structure of a multi-pass compiler	29
9.	Format of an IL element	30
10.	Block diagram of the ILLIAC IV TWS	32
11.	Block diagram of the brute-force ISL translator	51
12.	Flow diagram of the brute-force ISL translator	53
13.	Block structure of a typical ISL program	58
14.	Block structure of a translated ISL program	60
15.	Structure of the tables IDTAB, LKLIST and BLOCKSTACK	66
16.	Structure of the stack IFSTACK	67
17.	BIGTAB and MSTACK after an add column declaration	70
18.	Diagram of the files used by the ISL translator	87
19.	Structure of the file SEMANT	89
20.	Flow diagram of the selection of the input file	92
21.	Control decks for direct runs of the ISL translator	94
22.	The ZIP deck for the compilation of the compiler	102

LIST OF TABLES

Table		Page
1.	SET OF CHARACTERS WITH THEIR INTERNAL NUMBERS	21
2.	DOLLAR CARD CONTROL WORDS AND THEIR EFFECT.	23
3.	SCANNER CONTROL VARIABLES AND THEIR EFFECT	25
4.	THE TABLES IN TABLESF	35
5.	IL OPERATORS FOR DEMALGOL I	110

1. INTRODUCTION

The ILLIAC IV array computer presently under construction for the University of Illinois employs an advanced concept of parallel design to achieve a major increase in processing capacity. Obviously, in order to use such capacity efficiently, new procedure oriented languages must be developed to enable the user to take advantage of the parallelism of the system. As these languages are implemented it is only natural to expect that numerous modifications will need to be implemented as users become more experienced in the formulation of algorithms for an array computer. It would also be highly desirable that the users themselves have access to convenient methods of introducing small modifications in the basic languages so as to make them more suitable for a particular application. All these considerations pointed to a Translator Writing System (TWS) as a very convenient tool for achieving the objectives outlined above.

As a matter of convenience, this TWS will henceforth be referred to as the ILLIAC IV TWS. This does not imply, however, that the system is only suitable for generating compilers which produce ILLIAC IV machine code, although this has certainly been its main use to date. In fact, one of the main features of the system is to considerably simplify the task of producing a compiler for a given language L and machine M1 if a compiler has already been created, using the system, for L and another machine M2.

The ILLIAC IV TWS will run on the ILLIAC IV control computer, a Burroughs B-6500, and is presently being used on a Burroughs B-5500. The whole system is written in Burroughs extended ALGOL (henceforth called BeA). The reader is assumed to have a basic knowledge of this language as described in [1].

Briefly, a TWS is a system which, given the complete description of a programming language (i.e., its syntax and semantics) is capable of producing a compiler for that language. It is well known that a TWS can, in general, be divided into two parts:

a) A system, called the Syntax Preprocessor, which receives the formal syntactic description of a language and produces a recognizer for the well-formed constructs of such a language;

b) A system, called the Semantics Preprocessor, which receives the formal semantic description of a language and produces routines capable of generating the appropriate code for each valid construct in the language and of building descriptor tables.

As a result of this scheme, the compilers produced by TWS's can also be considered as composed of two parts:

a) the syntactic recognizer;

b) the semantic routines which describe data descriptor manipulation and code generation.

Obviously, there must be connections between these two parts; such connections receive the name of syntax-semantics links. Frequently the main syntax-semantics link consists of calls, in the syntactic recognizer, to appropriate routines in the package of semantic routines. Therefore, whenever a construct is accepted by the recognizer, a certain set of semantic routines (or semantic algorithms) is executed in order to

a) produce code corresponding to the construct recognized, and/or

b) do some housekeeping in the tables and stacks that keep track of what has been done so far.

Then the recognizer takes over again. Figure 1 illustrates the structure of the "classical" TWS outlined above.

This paper is devoted to the description of the semantics preprocessor of the ILLIAC IV TWS and its input, i.e., the semantic description it accepts. The syntax preprocessor is described by Beals [2] and the syntactic input by Mercer [3]. The system as a whole will be described by Machado and Northcote [4].

Some general considerations about the syntax and semantics of a programming language are in order. The following definitions have been presented:

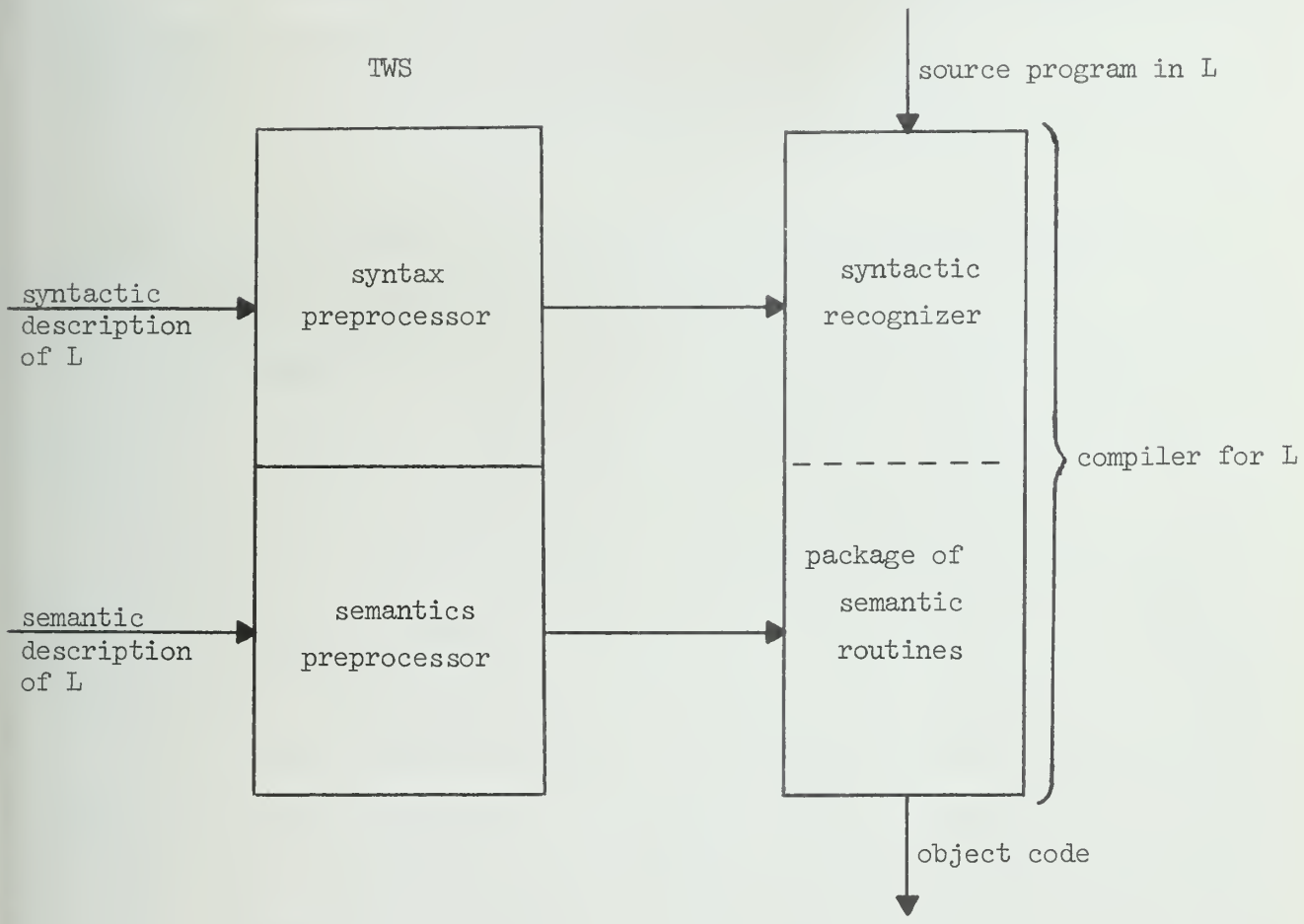


Figure 1. Classical structure of a TWS.

Syntax: "The syntax of a language is the study of its symbols and words *per se*. It is concerned with the structure of the well formed strings in the language and with the processes used for generating or recognizing such strings."

Semantics: "Semantics involves the assignment of meaning to the recognized syntactic constructs through the generation of descriptors for those constructs and the required object (output) string of symbols."

In short, it could be said that semantics is the part of a language associated with the *meaning* of the constructs whose *well-formedness* is checked by the syntax. A good example is provided by Naur [5] in section 2.4, in which the syntax and semantics of an identifier are defined for ALGOL-60. One immediately notices that the syntax is described in a formal notation, using Backus Naur Form (BNF) productions, while the semantics is presented in plain English. Since BNF is a formal system to describe syntax, a syntax preprocessor could be written to accept a BNF description of the syntax as its input. This is most reasonable since BNF productions describe syntax (or rather most of it) in a clear, concise, and precise way. Several other formal notations, besides BNF, are available to describe syntax and TWS's have been built using those notations as the syntactic language; for example, FPL (Floyd productions), TWINKLE (an extension of BNF used in the ILLIAC IV TWS), etc. The semantics, however, presents a problem. Obviously a description in English such as the one presented in the example mentioned above could not be accepted by a TWS so the need arises for a formal Semantic Description (SD) of a programming language. It can be seen immediately that the problem of a formal SD is much more complex than the syntactic description problem. Syntax (i.e., well-formedness) can be described in itself so it is possible to obtain a self-contained syntactic description. Semantics, on the other hand, is typically non-self-contained since it is related

with the relations between the constructs of a language and the constructs in other languages or entities in the real world. It is only possible to describe the meaning of something as a function of the meaning(s) of something else, so a set of primitive concepts or meanings must be postulated.

In the search for a good formal SD system, the following goals should be considered.

1) The SD must be simple enough to be conveniently utilized by the users of the TWS.
2) The SD should lead (directly, or through automatic transformation) to the generation of an efficient compiler.
3) It should lend itself to some type of syntax-semantics link.
4) If possible, it should be clear enough to be practically used to describe semantics, replacing the English descriptions.

Most of the formal systems for SD mentioned by Feldman and Gries [6] were reviewed and considered. Generally, SD's can be classified into two large groups: highly formal SD's and brute-force SD's. Highly formal SD's were basically developed for one of two purposes: precise specification of a programming language, avoiding the ambiguities of English, and for studying formal properties of programs such as equivalence of two programs. Examples of such highly formal SD's are: "verification conditions" as used by Floyd [7], Church's λ notation used by Landin [8], and the program schemata introduced by Ianov and discussed by Rutledge [9]. A highly formal SD would appear, a priori, to be the most convenient notation to describe semantics in a TWS since they are the counterpart of BNF, FPL and other highly formal syntactic descriptions successfully used in such systems. There is one crucial problem, however; goal number 2 above is presently impossible to attain with a highly formal SD. Since those approaches describe semantics basically using recursion over a small set of primitive concepts whose meanings have been postulated, they tend to generate compilers that would mimic this representation, i.e., the power of a modern digital computer

would be wasted simulating a great number of very elementary steps while the same computation could be performed using a few of the more sophisticated instructions available. Since the ILLIAC IV TWS is to be used to generate practical, operational compilers, efficiency is of paramount importance and the approach of picking a highly formal SD was abandoned.

This left, as an alternative, the use of a brute-force SD. This name refers to the fact that, in those SD's, the user himself must implement the semantics of the language by writing a package of semantic routines using a programming language. Such a language, called the semantic metalanguage, can either be a general purpose language like ALGOL, or a special language providing features to make the programming task somewhat easier. To this date most TWS's have adopted this kind of solution to the SD problem. The resulting compilers can be made reasonably efficient, depending on the ingenuity of the programmer writing the SD of the language.

A review of the existing systems, including systems that use a general purpose language as the semantic metalanguage (Booker and Morris [17], Trout [15],[16]) and the ones that use a special purpose metalanguage (Feldman [10], Northcote [11], [14], Iturriaga, et. al. [12], [13]), was carried out. With this background, it was decided to implement a special purpose semantic metalanguage for the ILLIAC IV TWS. This language, called ISL (for Illinois Semantic Language), is basically an extension of ALGOL. The extensions include some constructs often needed in the description of semantics, such as stack and table declaration and manipulation. Most of these constructs were suggested by Feldman's FSL (Formal Semantic Language) as described in [10]. The extensions also include control words to automate the use of the TWS to a significant extent

This approach seems to attain, within reasonable compromise, the four goals previously set forth. ISL, although a brute-force language, provides a relatively simple way for an experienced ALGOL programmer to describe semantics. Good efficiency can be obtained although this burden is mainly left to the user.

There are some very convenient forms of syntax-semantics link. Finally, despite the fact that it cannot completely replace an English description, a profusely commented ISL program can be a surprisingly clear, concise and precise semantic description for a user familiar with ISL.

An outline of the remainder of the paper follows. In Chapter 2 an overall description of the ILLIAC IV TWS is presented and is a background to the sequel. Chapters 3, 4 and 5 discuss the ISL implementation, give an informal description of ISL, and give details about the use of the ISL translator. The formal syntax of ISL can be found in Appendix A. Finally, Chapter 6 presents some general guidelines about describing semantics using ISL and a detailed discussion of an example: the ISL semantic description of DEMALGOL I, a small subset of ALGOL. The listings relative to this example are found in the remaining appendices.

For the reader who is not interested in details of implementation or use of the system, it is recommended that chapters 2, 4 and 6 only be read.

2. THE STRUCTURE OF THE TWS

This chapter describes the ILLIAC IV TWS. It is not, however, a manual for using the system since no details are given about how to actually run each program of the system. Such information, along with examples, can be found in [4]. The purpose of this chapter is simply to provide an understanding of the system as a whole since this background is needed in the discussion of some ISL constructs.

2.1 Basic Characteristics

The ILLIAC IV TWS is basically an implementation of the system proposed by Northcote [14]. Its main feature is the use of an intermediate language (IL) and multi-pass compilers. Although the system can also be easily used to generate one-pass compilers, its ability to generate multi-pass compilers is of primary importance in achieving conversion from one machine to another with a minimum of effort.

In a multi-pass compiler, the first pass translates the source program into IL code. The following passes use IL as both input and output so they merely transform the IL string. The final pass receives the last IL code string and outputs machine (or assembly) code. The IL code is syntactically very simple and can be parsed very easily. The basic number of passes for a compiler in this system is 2: the first pass transforms source language into IL and the second pass translates IL into machine code. Therefore, the first pass is machine independent and the second pass is language independent, which allows the production of $m \times n$ compilers (for m different languages and n different machines) by writing m passes 1 and n passes 2 instead of $m \times n$ one-pass compilers.

This basic 2-pass structure can be modified by adding intermediate passes between the first and last passes. Typically, intermediate passes will be optimization routines to transform the IL string into a more efficient

ordering. Since these optimizations might vary with the original source language and the destination machine, one can envision as a practical application a 4-pass compiler in which pass 2 would perform a language-dependent optimization and pass 3 a machine-dependent one.

2.1.1 Basic Block Diagram

Figure 2 illustrates the general structure of the ILLIAC IV TWS. Since the intermediate language has a fixed syntax, the IL recognizer does not have to be created for each language and it is simply an invariant package of procedures that is inserted only once in a multi-pass compiler and is shared by all passes after the first one. It should also be noted that Figure 2 presents only the principles of the system in a simplified way. Section 2.3 discusses the details of the actual implementation.

2.1.2 The Intermediate Language

The concept of multi-pass compilers with an IL works very well until it becomes necessary to define the operators in the IL. Then limitations and particularizations are practically inevitable and, when a new language or a new machine is being introduced in the system, one realizes that he needs a few more IL instructions to cope with some special features. In order to avoid this problem in the ILLIAC IV TWS, the IL is not fixed in content, i.e., it does not have a closed repertoire of instructions. What is fixed about the IL is its structure. In fact, the syntax of the IL can be defined simply as follows:

<IL program> ::= <IL operand> | <IL operator> | <IL program> <IL operand> | <IL program> <IL operator>

Thus an IL program is simply a string of elements, each of which is either an operator or an operand. Each IL element is basically a pair of

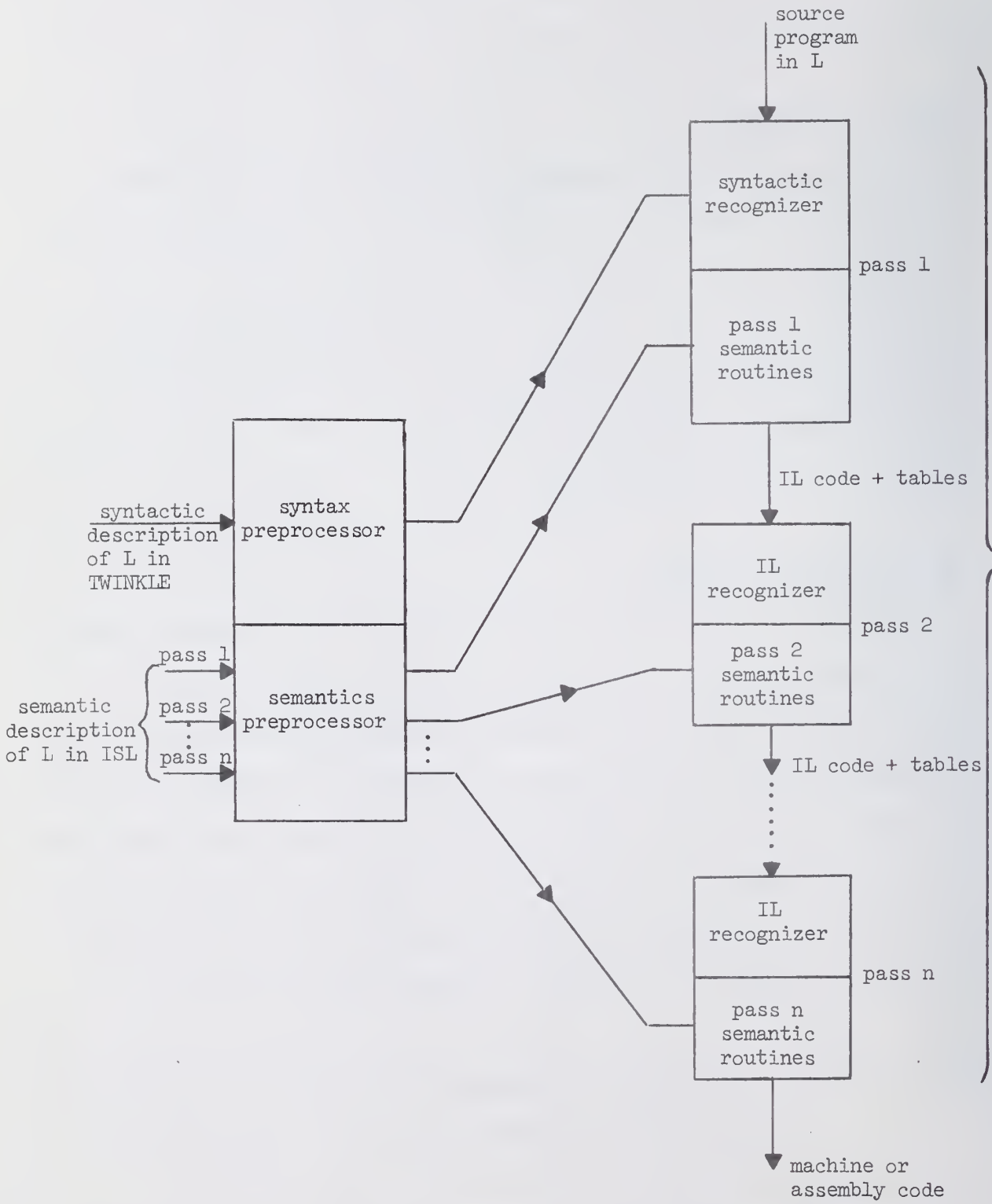


Figure 2. General structure of the ILLIAC IV TWS.

integers, the first integer is called the "table number" and the second the "entry". Any element with a non-zero table number is an operand; a table number equal to zero designates an operator. All the IL recognizer has to do to "parse" an element of the IL is to check the table number. If the table number is non-zero, the whole element is pushed into a stack; if the table number is zero, the semantic routine whose number is equal to the entry in that element is executed. Figure 3 is a block diagram of the IL recognizer. It is obvious that the IL string may, in fact, be Polish postfix notation and a semantic routine can pick its parameters, if any, by popping the stack. The fact that the set of operators is not fixed justifies calling such a language a "generalized intermediate language".

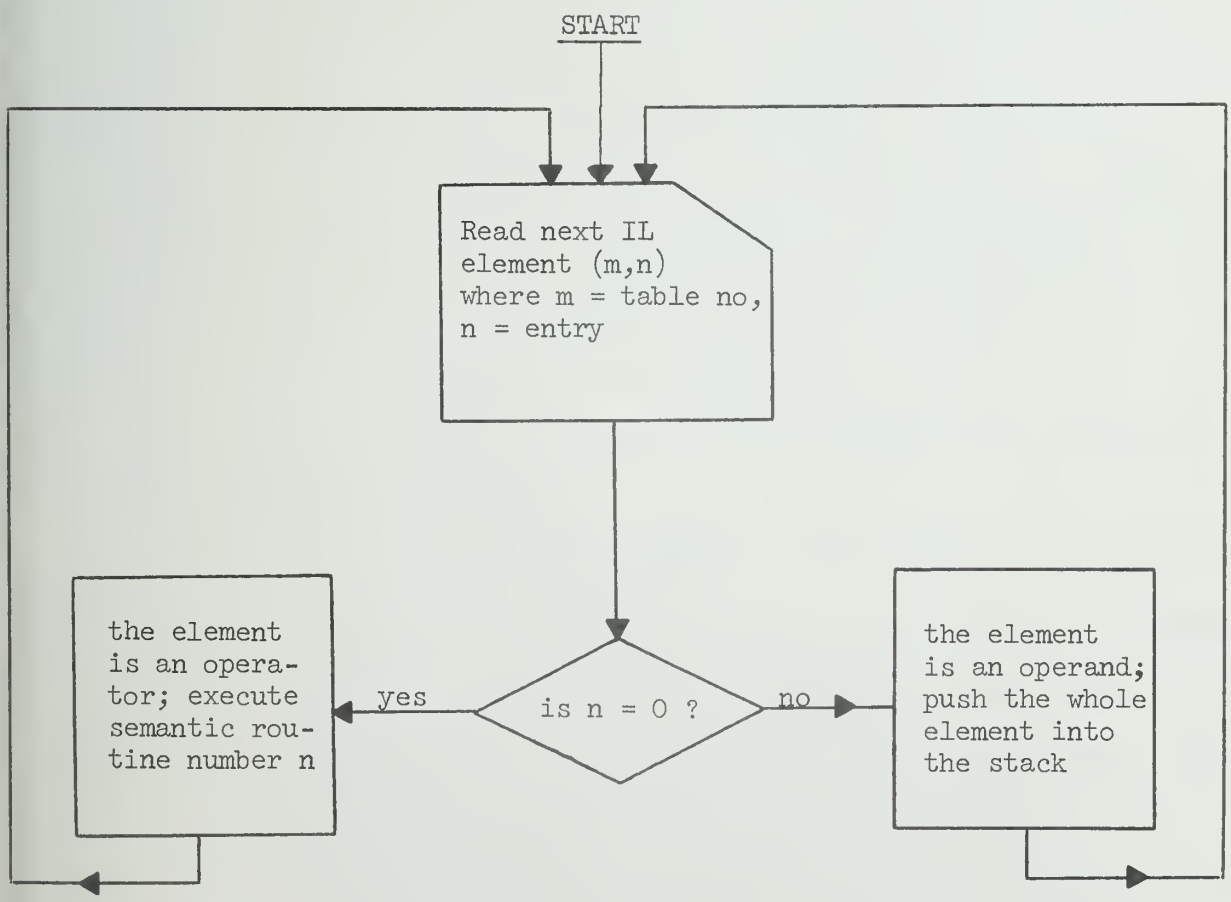


Figure 3. Block diagram of the IL recognizer.

Thus, pass i (i > 1) semantics is implemented with a modular approach; in a way, the user can add whatever operators he needs in the IL as soon as he also writes a semantic routine (i.e., an ISL procedure or block of code) specifying the new operator. This semantic routine will then be present in pass i and the new operator becomes available in the IL at that pass. After the first few compilers are written for a given machine M, there will be a good library of IL operators available along with the semantic routines defining them. The user will seldom have to define new IL operators as he will probably be able to describe his semantics using the operators already present in the library. However, he does have the possibility of adding new operators when they are needed to allow good efficiency of the generated code.

To give to the user the freedom (and, unfortunately, the burden) of defining his own IL operators means, in fact, that he has to decide how to divide his description of a language into passes. Considering the case of a 2-pass compiler, the user might either have pass 1 do almost all work and pass 2 would be a trivial emitter of machine code, or he could leave practically everything to be done in pass 2, pass 1 being only to recognize syntax and emit postfix code. Obviously, the correct approach is a compromise in which characteristics belonging to the language are translated in pass 1 and machine features are dealt with in pass 2. It is interesting to note, however, that pass 1 should fragment the input program into instructions not simpler than the instructions available in the machines for which a pass 2 will be written. For instance, if pass 1 translates the statement A $\oplus$ B (where $\oplus$ means exclusive or) into the following intermediate code: A, NOT, B, AND, A, B, NOT, AND, OR, then every machine will execute the statement in this way, which is inefficient for a machine that has an exclusive or instruction in its repertoire. These considerations about where to divide a compiler between passes have to be loose, since only general guidelines may be given. It is hoped that the example in Chapter 6 will provide some additional insight with respect to this problem.

2.2 The Structure of Translators Generated by the System

The right hand side part of Figure 2 contains a simplified block diagram of a translator generated by the ILLIAC IV TWS. Each pass is basically composed of a syntactic recognizer and a package of semantic routines. The format of each package is essentially the same for every pass. The syntactic recognizer, however, is completely different for pass 1 and for the subsequent passes. All passes i ($i > 1$) use the same IL recognizer, which is quite simple. Pass 1, however, needs a very elaborate parser and scanner in the syntactic recognizer. For this reason, the case of a one-pass compiler will be studied separately with its syntactic recognizer described in detail. Then the more complex multi-pass compilers can be explained without further comments about their first passes. It should also be noticed that the primary output produced by the TWS system is not a final compiler but a BeA program, i.e., the compiler in source code form. The ALGOL compiler is then used to "compile the compiler" and the resulting machine code is the final compiler. Therefore, discussion of the compilers generated will be in terms of such compilers as ALGOL programs. Reference to the ALGOL block structure of the compilers will be made quite often to clarify the scope of each procedure or table in the compiler.

2.2.1 One-pass Compilers

Figure 4 presents a simplified ALGOL block structure of a one-pass compiler generated by the ILLIAC IV TWS. The main components of the syntactic recognizer are two packages of declarations: the SCANNER package and the PARSER package. Between those two groups of declarations there is a single executable statement: a call to procedure INITIALIZE whose basic role is to fill the scanner tables (the scanner is table-driven). For reasons that will become apparent in section 2.2.2, a block of declarations is detached from the SCANNER package and placed at the very beginning of the program. It is called the GLOBAL package; among its contents are: NUMERRS, which contains the number of errors detected

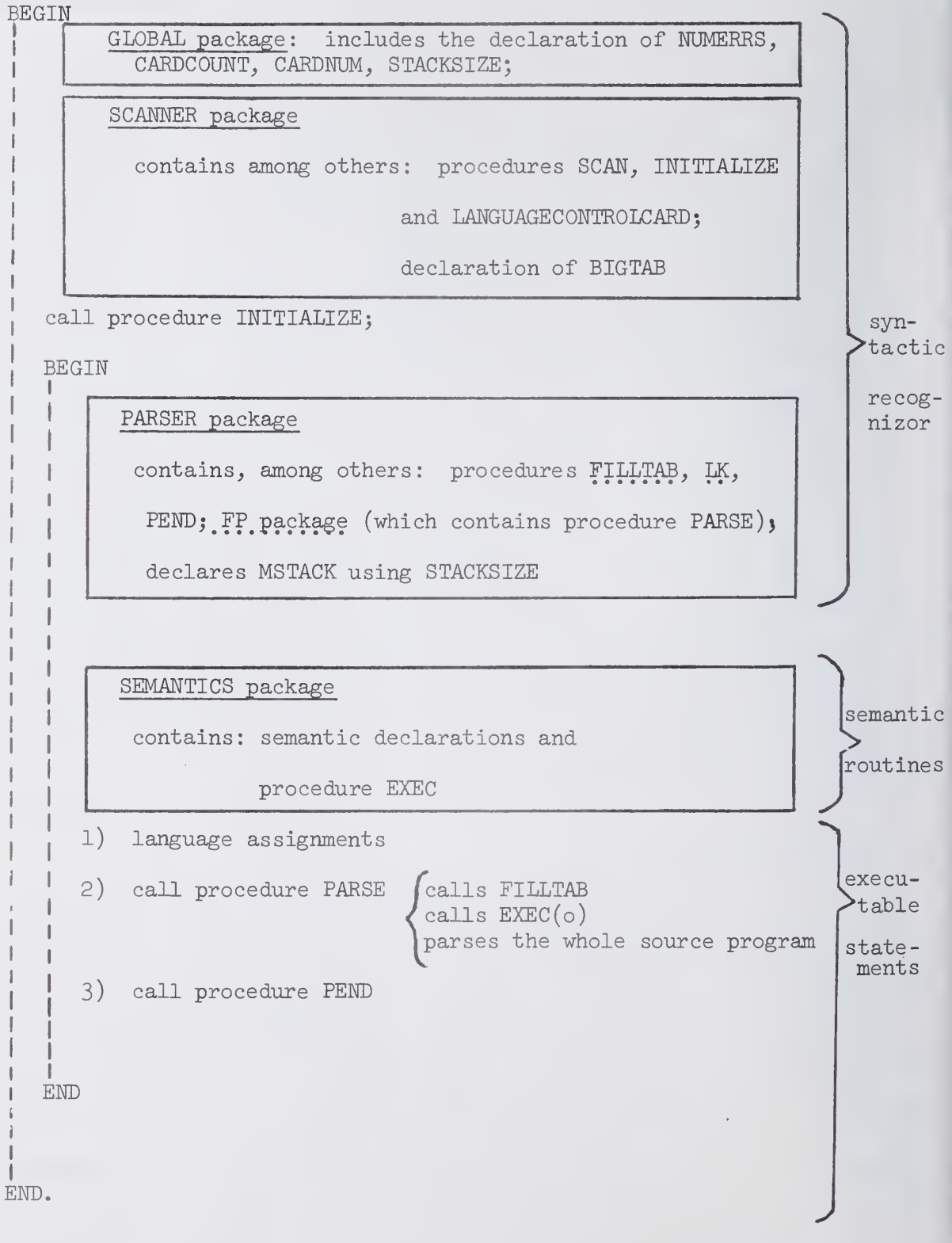


Figure 4. ALGOL block structure of a one-pass compiler.

during compilation; CARDCOUNT, which contains the number of cards compiled; CARDNUM, which contains the number of the sequence field (columns 73-80) of the card presently being compiled; STACKSIZE, which contains the number of locations desired in the main syntactic recognition stack. The fact that these four quantities are always the first four declarations in a compiler allows them to be easily monitored with console commands whenever the compiler is running in a B-5500. Following the PARSER comes the SEMANTICS package which contains the declarations of the semantic variables and routines. Finally, there are a few executable statements. The most important of these is a call to procedure PARSE which results in compilation of the source program. Then procedure PEND (for parser end) is executed to get some statistics about the compilation and the execution of the compiler is terminated. Each block will now be analyzed in more detail.

2.2.1.1 The Scanner and BIGTAB

The main procedure in the SCANNER package is an alpha procedure called SCAN. Each time the parser needs another element from the input string (source program), it calls SCAN which scans the input and returns the next element; it includes the reading of new cards as necessary. An updated version of the scanner, to be described by Baker, will be inserted in the system as soon as it is operational.

The scanner recognizes basically four types of elements, each designated by a type number:

type 1 : Identifiers (as defined in [1], pages 2-5)

type 2 : Numbers (as defined in [1], pages 2-6)

type 3 : Strings (as defined in [1], pages 2-7)

type 15: Terminal symbols. These can appear in three different forms:

a) Single characters.

Example: ← (the single character left arrow)

b) Reserved words. These are identifiers that appear as terminal symbols in the syntactic definition of the language; the scanner distinguishes them from common identifiers.

Example: BEGIN (in ALGOL)

c) Special words. This is a special mode in which the user must immediately precede all identifiers which are terminals with a special character (usually #).

Example: #BEGIN (in ALGOL)

Since all terminal identifiers are identified by the special prefix, the same identifier could appear without the prefix and would be considered a valid identifier. This is one advantage of the special word option. Another is the fact that scanning identifiers is somewhat faster, since all the reservedwords do not have to be matched against each identifier as happens in the reserved word option. The obvious disadvantage is the use of the prefix before each terminal identifier. The same compiler can be used either with reserved words or with special words, dependent on a control card option introduced at compile time.

The unallocated symbol type numbers may be utilized by the user to define additional symbol types for his language.

As the scanner first recognizes identifiers, strings and numbers, they are entered in a table called BIGTAB. When any of these elements appears a second time the address of the previous entry is returned. BIGTAB is a linear table of about 8000 48-bit words. Each word in BIGTAB can be one of three types:

a) heading word;

b) character word;

c) number word.

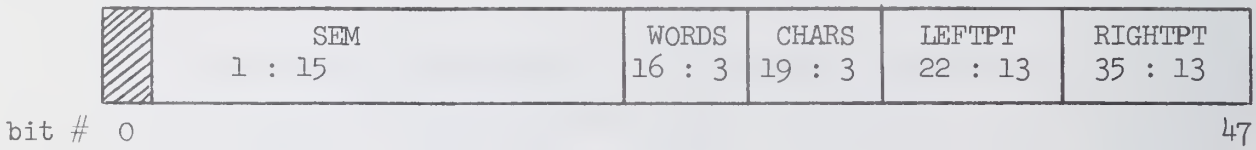
Numbers are stored using the standard B-5500 representation with a mantissa and an exponent (which is zero for integers). The formats of heading words and character words are given in Figure 5. Each set of elements of the same type is entered in BIGTAB according to a binary tree structure with left and right pointers linking all its elements thus enabling the scanner to perform fast table lookups. This structure allows several different trees to be interleaved in BIGTAB, each tree having a different root. The first locations of BIGTAB are used as follows: BIGTAB [i] ($0 \leq i \leq 15$) contains a pointer to the root of the tree of elements of type i. Thus BIGTAB [1] points to the root of the tree of identifiers. The first location of BIGTAB used for an entry is 16 and the locations are used in increasing sequence.

Entries in BIGTAB for each type are specified as follows.

identifiers: Each entry contains one heading word and from one to eight character words (as needed) containing the BCD for the identifier. Thus an identifier in this system may be up to 48 characters long. Figure 6 illustrates the entry of an identifier in BIGTAB. Notice that the address of an entry in BIGTAB is always the address of its heading. The number of words used per entry may vary from 2-9. Unused characters in the last word of the entry are filled with blanks.

numbers: Each entry contains one heading word and one number word in which the value of the number is stored in the conventional B-5500 way. The field SEM in this case is set to a number that represents the sum of all words used so far in entries of numbers and strings. This is useful when it is desired,

Heading word:



SEM: Semantics field; reserved for use by the semantics

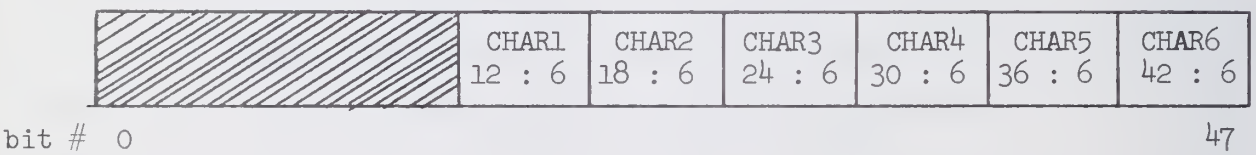
WORDS: Contains n-1 where n is the number of extra words (besides the heading) used for this entry

CHARS: Contains the number of characters in the last word of this entry

LEFTPT: Left pointer; contains the address of the heading of the entry to the left of this one in a binary tree structure

RIGHTPT: Right pointer; contains the address of the heading of the entry to the right of this one in a binary tree structure

Character word:



Each field contains one character in the Burroughs set of 64 characters, according to the conventional integer assigned to each character in Table 1.

Figure 5. Format of words in BIGTAB.

identifier: IMPLEMENTATIONS

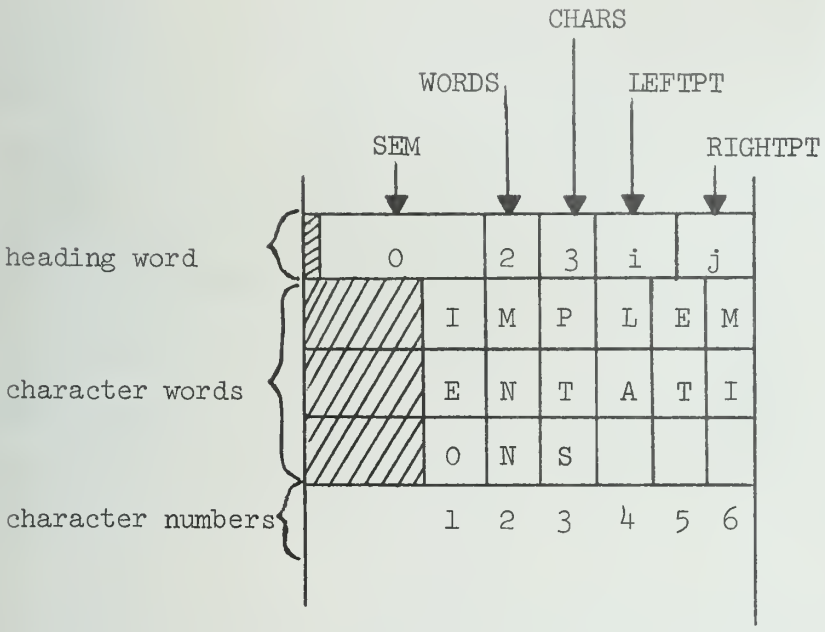


Figure 6. Sample entry of an identifier in BIGTAB.

at the end of pass 1, to extract all numbers and string entries out of BIGTAB and condense them in a table of literals to be passed to pass 2.

strings: Are stored similarly to identifiers, with the limiting quotes removed. The SEM field is set as described above for numbers.

special or reserved words: Are stored similarly to identifiers except that in the field SEM there is a code number. This number is assigned to each reserved word by the syntactic preprocessor. The first reserved word found is assigned the value 66 and so on in sequence. Thus each terminal has a number assigned to it: if it is a single character, such number is the character number as defined in Table 1. If it is a multi-character terminal, it has a code number ≥ 66 assigned by the syntactic preprocessor.

Note that the numbers 64 and 65 were left out. They are used to indicate "end-of-file" and "illegal terminal",respectively. End-of-file is the terminal returned when there are no more elements to be read in the input string. Illegal terminal is returned when, under special word option, a special word is found which had not been defined in the syntactic specifications. Under the special word option there are, in general, four trees in BIGTAB: identifiers, numbers, strings and special words. Under reserved words, however, the tree of special words does not appear as an independent tree but as the initial part of the identifier tree. If an "identifier" falls within this initial part it is, in reality, a reserved word; otherwise it is a proper identifier.

Whenever the scanner is called, it returns essentially a pair of integers: (n,m) where n is the type and m is the entry. In all cases, n will be the type of the element scanned (i.e., 1, 2, 3, or 15 accordingly as the element scanned was an identifier, etc.). The setting of m depends on the type. For

TABLE 1

SET OF CHARACTERS WITH THEIR INTERNAL NUMBERS

character	=	number	character	=	number	character	=	number	character	=	number
0	=	0	+	=	16	×	=	32		=	48
1	=	1	A	=	17	J	=	33	/	=	49
2	=	2	B	=	18	K	=	34	S	=	50
3	=	3	C	=	19	L	=	35	T	=	51
4	=	4	D	=	20	M	=	36	U	=	52
5	=	5	E	=	21	N	=	37	V	=	53
6	=	6	F	=	22	O	=	38	W	=	54
7	=	7	G	=	23	P	=	39	X	=	55
8	=	8	H	=	24	Q	=	40	Y	=	56
9	=	9	I	=	25	R	=	41	Z	=	57
*	=	10	.	=	26	$	=	42	,	=	58
@	=	11	[	=	27	*	=	43	%	=	59
?	=	12	&	=	28	-	=	44	≠	=	60
!	=	13	(	=	29	)	=	45	≡	=	61
>	=	14	<	=	30	;	=	46	]	=	62
≥	=	15	←	=	31	≤	=	47	"	=	63

any type not equal to 15, m is the address of the entry of the element in BIGTAB. For type 15, m is the code number associated with the terminal, as explained previously.

To complete the description of the scanner it is necessary to describe the special meaning of the characters "%" and "$". The character "%", when not embedded in a string, terminates the reading of a card causing all the remaining characters in that card to be ignored. The character "$", when in column 1 of the card and not embedded in a string, causes the card to be interpreted as a control card. The card is expected to contain a string of control words separated by blanks; the string must be terminated by a semicolon (";"). Table 2 presents a list of the available control words and their effect. For certain applications, the user may desire to add some control words of his own, besides the ones provided by the system. One of the procedures in the SCANNER package, LANGUAGECONTROLCARD, allows this facility. The user is expected, in this case, to write his own procedure LANGUAGECONTROLCARD to accept the additional options and insert it in place of the original procedure, which is simply a marker and has no action whatsoever.

The way the scanner treats blanks and the way it recognizes multi-character elements can also be modified to a great extent by setting certain variables to the appropriate values. Table 3 presents the complete list of such scanner control variables and their effect. There is, however, one important restriction in the use of these variables: when a semantic routine modifies a scanner control variable, one must be sure that the element which must be scanned under the new mode has not been scanned yet. Such could be the case if look-ahead is needed at that point in the parsing.

TABLE 2

DOLLAR CARD CONTROL WORDS AND THEIR EFFECT

CONTROL WORD	obs	AFFECTS	EFFECT
LIST		scanner	Prints the source cards on the line printer as they are needed. (default)
NOLIST or NLST		scanner	Turns off option above.
RSWORD * or RSWD		scanner	Uses reserved words.
SPWORD * or SPWD	must be followed by a special character: c	scanner	Uses special words marked by the prefix c (default with c = "#").
STACK *	must be followed by an unsigned integer: n	parser	Uses a parsing stack n words long (default: n = 100).
DEBUG or DBUG		parser	Requests all debugging aids available; is equivalent to setting the next six options.
NODEBUG or NBUG		parser	Turns off option above (default).
EXEC		parser	Prints message as each semantic routine is called for execution.
NOEXEC or NEXC		parser	Turns off option above (default).
SEEKNT		parser	Prints message when the parser starts to look for each specific non-terminal.
NOSEEK		parser	Turns off option above (default).
FINDNT		parser	Prints message when the parser finds the non-terminal it was looking for.
NOFINDNT		parser	Turns off option above (default).
STACKLIST		parser	Prints the situation of the stack (up to the top ten symbols only) as each terminal is pushed into it.
NOSTACKLIST		parser	Turns off option above (default).

TABLE 2

(CONTINUED)

CONTROL WORD	Obs	AFFECTS	EFFECT
PARSER		parser	Prints the number of the first FP in a group as each new group of FP's is initiated.
NOPARSER		parser	Turns off option above (default).
SCAN		scanner	Prints message as each source program element is scanned.
NOSCAN		scanner	Turns off option above (default).
TIMES		PEND	At the end of the compilation, prints times spent in each activity (scanning, parsing, etc.).
ILCARD		IL	Outputs in the IL string information about the card number each time a new card is read.
ILLIST		IL	Starts printing the IL string as it is read by pass 2; this automatically turns on the option above.
NOILLIST		IL	Turns off option above (default).

* These options are valid only when input before the first non-control card in the source deck.

TABLE 3

SCANNER CONTROL VARIABLES AND THEIR EFFECT

VARIABLE	TYPE	INITIALIZED TO	VALUE	EFFECT
SCANMODE	INTEGER	0	0	Recognizes all types of elements* (id's, numbers, strings, etc.); blanks are completely ignored.
			1	Same as 0 except blanks are recognized; multiple blanks are reduced to a single blank.
			2	Does not recognize ids, etc. as as such but as a string of single characters. Blanks are ignored.
			3	Same as 2 except blanks are also returned; multiple blanks are reduced to a single blank.
			4	Same as 0 but does not make any entry in BIGTAB.
			5	Same as 3 but multiple blanks are not reduced to a single blank.
FRSTCOL	INTEGER	1	i	Starts reading each card at column i.
LASTCOL	INTEGER	72	i	Stops reading each card after column i.
LETTERCHAR	BOOLEAN	FALSE		If true, recognizes identifiers character by character.
DIGITCHAR	BOOLEAN	FALSE		If true, recognizes numbers character by character.
STRINGCHAR	BOOLEAN	FALSE		If true, recognizes strings character by character.
SPECIALCHAR	BOOLEAN	FALSE		If true, recognizes special words character by character.

* conditional on the setting of the four boolean variables: LETTERCHAR, DIGITCHAR, STRINCHAR and SPECIALCHAR

2.2.1.2 The Parser and MSTACK

The PARSER package contains all the procedures that actually parse the input string of source code. Since the final result of the syntactic preprocessor is a deterministic set of Floyd Productions (FP's) describing the syntax, the task of the PARSER procedures is to apply those productions to the source string. This is done by the FP package. Also important are the procedures LK, which does the lookahead, and FILLTAB, which is executed at the beginning of the compilation and initializes several tables, among them BIGTAB with the reserved words. The parser uses a stack called MSTACK whose word format is illustrated in Figure 7. MSTACK contains a number of 48-bit words equal to the value of STACKSIZE. This number can be set with a control option (see Table 2); the default size is 100.

As the elements of the source string are scanned, they are pushed into MSTACK as described in Figure 7. When the contents of the top n locations of MSTACK matches the n-symbol right-hand side of a production, the parser may decide (depending perhaps on the lookahead situation) to effect a reduction. The reduction then consists of discarding the top n-1 symbols in MSTACK. The n-th symbol, now top of MSTACK, is associated with the nonterminal on the left hand side of the production which was applied. Note that this association is only internal, however, and physically the contents of MSTACK remain as they were.

Example: Assume that MSTACK is: . . . #GO #TO| (the vertical bar is used to indicate the top of the stack) and that the BNF production: <GO TO SYMBOL> ::= #GO #TO will be applied. The position of MSTACK containing the #TO is erased and all information stored in it is lost. The top of MSTACK is now associated with the nonterminal <GO TO SYMBOL> but the contents of the top of MSTACK continues to be: (type 15, entry = number associated with the terminal #GO). Technically, one could say that the actual contents of a position of MSTACK associated with a nonterminal are the leftmost terminal of the string of terminals that were reduced to that nonterminal.

MSTACK word:

LTYPE: In pass 1, contains a copy of the five rightmost bits of the TYPE field; in pass i (i > 1), this field can be used for semantics.

SEM: Semantics field, reserved for use by the semantics. In pass 1, when the result of a scan is placed in MSTACK, the rightmost 12 bits of the SEM field in BIGTAB are automatically copied in the SEM field of MSTACK.

ENTRY: In pass 1, contains the number of the element, as returned by the scanner; in pass i (i > 1), this field can be used for semantics.

TYPE: In pass 1, contains the type number of the element, as returned by the scanner; in pass i (i > 1), this field contains the table number of an IL operand.

ILENTRY: In pass 1, when the result of a scan is placed in MSTACK, this field is initialized to zero unless the element is a terminal in which case the code number is entered; in pass i (i > 1), this field contains the entry number of an IL operand.

Figure 7. Format of the words in MSTACK.

2.2.1.3 The Semantic Routines

Much will be said in subsequent chapters about the SEMANTICS package. For the time being, it is only important to note that the main procedure of the SEMANTICS package is called EXEC. EXEC has one integer parameter and the result of a call on EXEC with parameter n is that semantic routine number n (only) will be executed. Thus, when the parser decides it is time to execute semantic routine number (say) 43, all it does is to call EXEC (43).

2.2.2 Multi-pass Compilers - the IL Recognizer

Figure 8 presents a simplified ALGOL block structure of a multi-pass compiler generated by the ILLIAC IV TWS. It should be noted initially that the GLOBAL package has been moved to the very beginning of the compiler. Therefore, the declarations it contains are global to all passes and may be used by all. Thus GLOBAL includes all the declarations of elements that the system must use in all passes. Notice also that STACKSIZE is now global to the whole program so all passes will use it as the size of their MSTACK. Since all passes i ($i > 1$) use the same IL recognizer, the procedures in the IL RECOGNIZER package are entered only once and are accessed by all those passes. It should also be noted that one more package was introduced: the ALGOL-HEAD package. It can be considered the user's counterpart of the GLØBAL package; it contains the declarations of all the semantic elements that should be transferred between passes. Of course, the main vehicle of transfer of information between passes should be the IL string. However, it is also very often necessary to transfer some tables and stacks to which IL operands point. In fact, due to the small size (16 bits) of an intermediate language operand, unless the language being implemented is quite simple, such operands cannot contain the whole information to be communicated between passes. It is up to the user to decide which information he wants transferred between passes and to place the declarations of those elements in HEAD instead of in a particular SEMANTIC package.

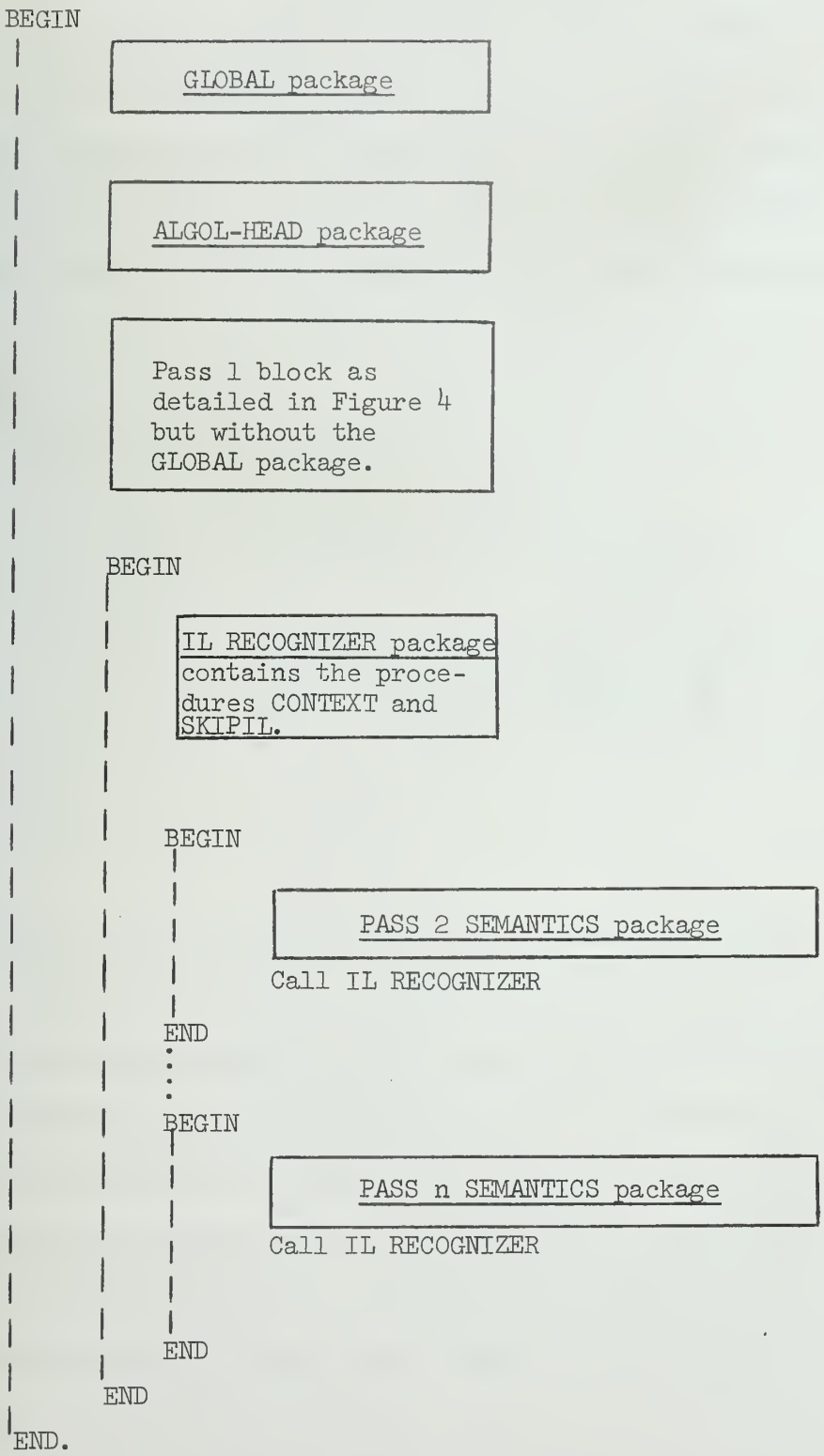


Figure 8. ALGOL block structure of a multi-pass compiler.

The IL structure and its recognizer have already been generally described in section 2.1.2. All that is left is to add some implementation details. The format of an IL element is shown in Figure 9. Four bits are reserved for the table number and 12 bits for the entry. In this way, three IL elements can be packed in one B-6500 48-bit word. This is not being done with the B-5500 because one of the 48 bits is inaccessible to the user. The stack used by the IL recognizer is also called MSTACK and the use of each field is described in Figure 7.

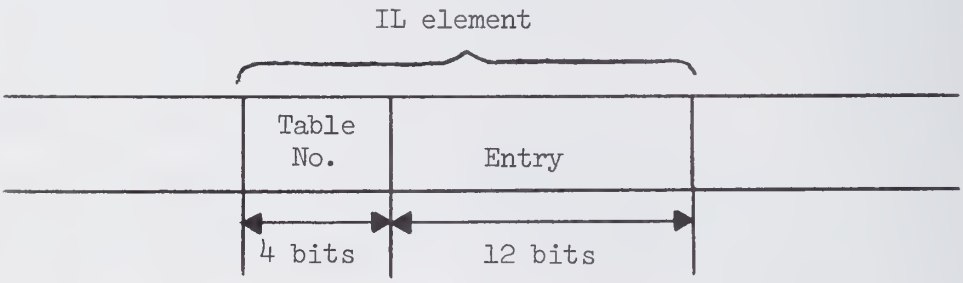


Figure 9. Format of an IL element.

Finally, it should be added that although the ideal form of the IL is pure postfix notation, sometimes it is convenient to give the user freedom to scan the IL string from inside a semantic routine for a lookahead, a local optimization, or some other trick. For this purpose two procedures are available: CONTEXT (n) which is typed integer and returns the element in relative position n in the IL string ($-30 \leq n \leq 59$) and SKIPIL (n), a non-typed procedure which displaces the IL string by n where n is any integer. Two IL operators have a predefined meaning for the IL Recognizer. Operator number 64 is the end-of-file transfer. When it is recognized, the Recognizer stops and the compiler proceeds to the next pass (if any). A semantic routine for operator number 65 is provided

by the Recognizer itself. When that operator appears in the IL string it is known that the IL ENTRY field on top of MSTACK contains a card number and TYPE contains either 2 (meaning list IL) or 1 (meaning do not list IL).

In the first case the routine associated with operator 65 prints a card number message on the line printer; in the second case nothing is done. Thus the user may obtain at pass 2 a listing of the IL string being processed along with the numbers of the cards that originated it. This capability is controlled by the three control words that affect IL (see Table 2). Obviously, the user may choose to supply his own semantic routine for operator 65 if he wants something else done at card boundaries during the reading of the IL string.

2.3 The Actual System

This section contains a detailed description of the block structure of the ILLIAC IV TWS as it has been actually implemented. The function of each block is briefly discussed with references to the paper in which they have been fully described.

2.3.1 Detailed Block Diagram

Figure 10 presents the detailed block diagram of the ILLIAC IV TWS. The actual system is composed of five major programs and a TWS Symbolic Library containing several pieces of ALGOL source code. In Figure 10 the upper row, containing four boxes, corresponds to the syntax preprocessor. The approach of dividing the syntax preprocessor into four programs presents two major advantages. If the whole syntactic preprocessor had been packed in only one program, the resulting code would be extremely large and could not be run efficiently in the B-5500 due to large overlay time. Secondly, each of the four programs performs a distinct logical task. The B-5500 system provides facilities for initiating the execution (on a multiprocessing basis) of one program by means of executing a special statement in another program being executed. Therefore,

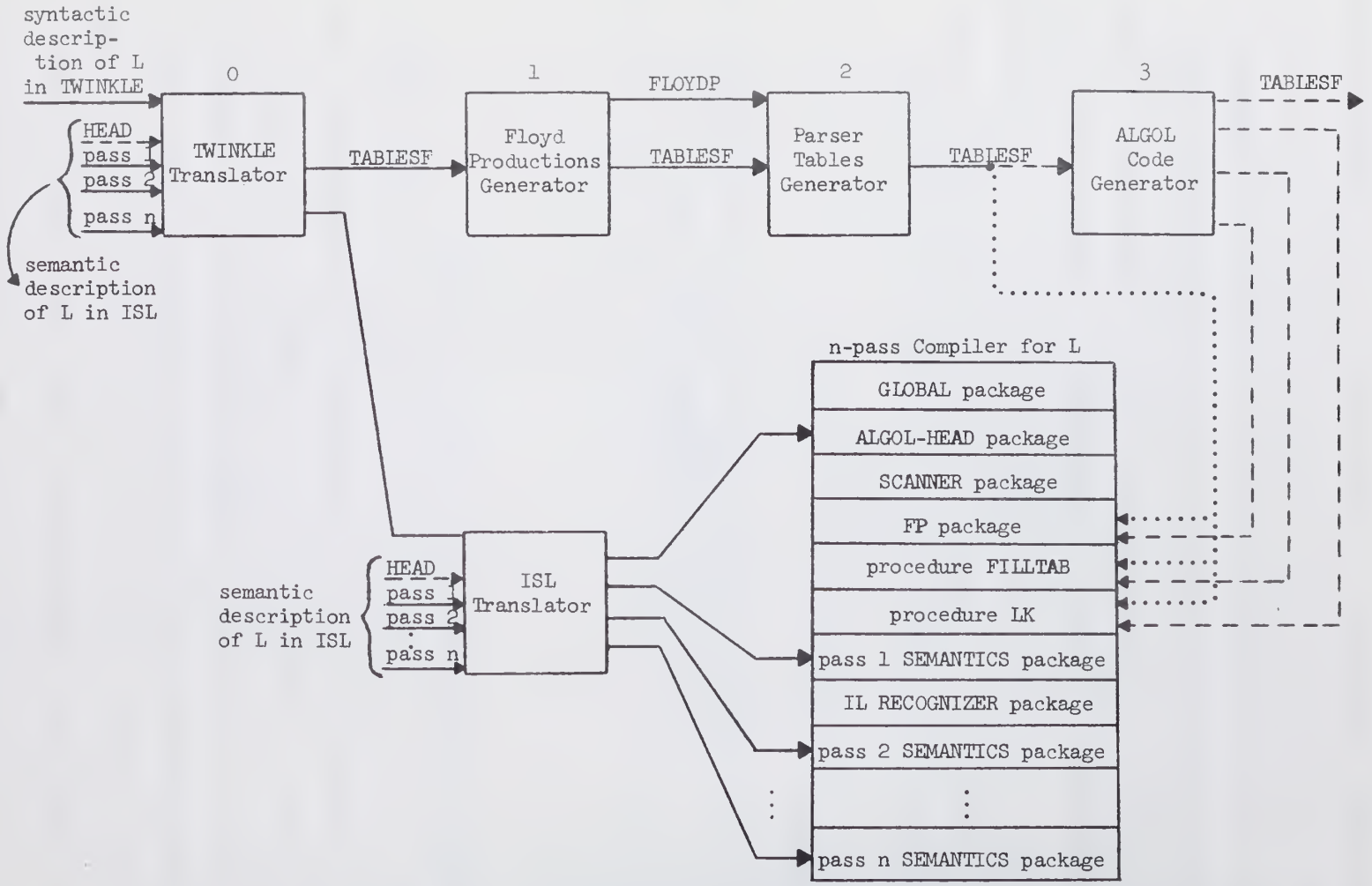


Figure 10. Block diagram of the ILLIAC IV TWS.

dividing the syntax preprocessor does not pose the disadvantage of requiring the user to execute four programs instead of only one. The user executes only the first program (the TWINKLE Translator) and each program, just before ending, initiates the execution of the next one. In fact, it is possible to input to the TWINKLE Translator both the syntactic and semantic description of a language L and have the system, completely automatically, end up with a compiler for L (in executable form). This involves:

a) the execution of the four programs of the syntax preprocessor;

b) the execution of the ISL translator which is the only program of the semantics preprocessor;

c) "mounting" the ALGOL source for the compiler by putting together several pieces of ALGOL code; some pieces are generated as the end result of the syntax preprocessor, others are generated by the ISL translator, and some are taken from the TWS Symbolic Library;

d) executing the ALGOL compiler on the ALGOL source program obtained in (c) in order to get the compiler for L into machine code.

The TWS Symbolic Library contains, in ALGOL form, all the pieces of the system which are not language-dependent. They are basically the GLOBAL package, the SCANNER package and the IL RECOGNIZER package. The Library is kept on B-5500 disk and its routines are available for insertion in each compiler generated by the system. The block representation of the compiler in Figure 10 shows clearly which pieces are generated by the syntax preprocessor and which are generated by the ISL translator. The remaining pieces come from the TWS Symbolic Library. Each of the five programs in the system will now be discussed separately.

2.3.2 TWINKLE and the Syntax-Semantics Links

The first program of the syntax preprocessor and initial program of the whole system is the TWINKLE translator. Its main task is to translate the syntactic description from its original TWINKLE form into BNF in tabular form. TWINKLE, the syntactic metalanguage used by the system, is fully described by Mercer [3]. Besides accepting the classical BNF productions, TWINKLE allows a number of other constructs such as the "Kleene star" and English-like productions that make the syntactic description very clear and self-documenting. Appendices B and C present TWINKLE descriptions of a small subset of ALGOL written in two different ways. In Appendix B, only BNF productions were used while in Appendix C full advantage was taken of the English-like constructs. This language, called DEMALGOL I, is used in Chapter 6 to provide an example of a semantics description using ISL.

The output of TWINKLE is a file called TABLESF which contains a number of tables. This file is left on the B-5500 disk and is a common pool of information for all the programs of the system, i.e., the subsequent programs use the information in TABLESF as input and/or add some more tables to it. A full discussion of each table in TABLESF is beyond the scope of this paper but a list of these tables with a summary description of what they contain is presented in Table 4. The TWINKLE translator produces another output file besides TABLESF; it is called ACTIONS and contains information about the semantic action calls that appeared in the syntactic description. This information is then used by the ISL translator to match the semantic actions (or routines) definitions in the semantic description with their calls in the syntax. Finally, it should be noted that in Figure 10 the semantic description appears as input, with dashed lines, to two different programs: the TWINKLE translator and the ISL translator. This indicates that the semantic description may be input to either of these programs. The regular way is to input it directly to the ISL translator. However, if desired, the semantic description may be appended to the syntactic

TABLE 4

THE TABLES IN TABLESF

TABLE NAME	CREATED BY*	MODIFIED BY*	USED BY*	DESCRIPTION
TINDEX	0		2,3,C	Terminal indexes: Contains, for each terminal, the address of the entry of the terminal in SPSTAB.
NTINDEX	0		2,3,C	Nonterminal indexes: Contains, for each nonterminal, the address of the entry of the nonterminal in NONTAB.
OPRINDEX	0		2,3,C	Operator indexes: Contains, for each semantic action name, the address of the entry of the semantic action name in OPRTAB.
SPSTAB	0		2,3,C	Special Symbol Table: Contains, in a binary tree structure, all the terminals of the language. Used to initialize BIGTAB.
NONTAB	0		2,3,C	Nonterminals Table: Contains, in a binary tree structure, all the nonterminals of the language.
OPRTAB	0		2,3,C	Operator Table: Contains, in a binary tree structure, all the semantic action names used in the language.
PROTAB	0		1	Production Table: Contains, in a coded form, the BNF productions that describe the syntax.
PATTERN	0	1,2	all	Patterns Table: Contains strings of bits that allow efficient implementation of multiple lookaheads, stack tests and the implementation of the "any symbol" feature.
NTODO	1		2	Need to do Table: Contains flags indicating which of the Floyd Productions are actually needed for parsing.
COMGPS	1	2	2,3,C	Combined groups Table: Contains the combined groups of Floyd Productions. See [2] for an explanation.
ERRTAB	1		3, C	Error Table: Contains, for each nonterminal, a list of all the terminals that can follow it. It is used to recover from errors.

TABLE 4

(CONTINUED)

TABLE NAME	CREATED BY	MODIFIED BY	USED BY	DESCRIPTION
GROUPTYPE				Grouptype Name Table: Contains, for each group type, its BCD mnemonic. (Not used at present. Needed for future additions to the system.)
FPTAB	2	3	3,C	Floyd Productions Table: Contains sets of triples which include all the information needed for the implementation of a Floyd Production.
STKTAB				Stack Test Table: Contains strings of symbols for performing stack tests for more than one symbol. (Not used at present. Needed for future additions to the system.)
LKTAB	2		3,C	Lookahead Table: Contains strings of micro-instructions for implementing multi-symbol look-ahead tests.
SEMTAB	2		3,C	Sequential Semantic Calls Table: Contains information needed to perform two or more semantic action calls in a row in a single FP.
XBTAB	2		3,C	Dynamic Transfers Table: Contains information needed to perform a dynamic transfer by the parser.
GROUPNAME	2		3	Group Name Table: Contains information regarding the mnemonic name associated with each group.
GROUPPTR	2		3	Group Pointer Table: Contains, for each group, a pointer to its starting location in FPTAB.
EXCOMGPS	3		C	Executable FP's Combined Groups Table: Same as COMGPS for use with executable FP's.

*The program numbers mean: 0 - TWINKLE Translator, 1 - Floyd Production Generator, 2 - Parser Tables Generator, 3 - ALGOL Code Generator, C - the' resulting COMPILER, all - all five above.

description and be input to the TWINKLE translator. This program will separate the two descriptions and insert the semantics in the file ACTIONS, along with the information that is normally there. The TWINKLE compiler will then initiate the execution of both the FP generator and the ISL translator. At the end of the process, the ISL translator or the ALGOL code generator (whichever ends last) assembles all the pieces of the compiler and initiates the execution of the ALGOL compiler to compile the required compiler. Thus it is possible to obtain the final compiler simply by executing the TWINKLE compiler (with the proper options set) and presenting to it both the syntactic and semantic descriptions. The different types of syntax-semantics links present in the system will now be discussed.

2.3.2.1 Semantic Action Calls

The main syntax-semantic link in the system is the semantic action call. It has already been established that the syntactic description contains BNF productions (or generalized forms of BNF productions) and the semantic description contains semantic routines (the term "semantic action" is used interchangeably with "semantic routine"). The link between these two is provided by semantic action calls of the type: @S <identifier> placed anywhere (except at the beginning) in a BNF production, meaning that when the parser reaches precisely that point in a production control must be given to the semantic action labelled by the <identifier> . After the execution of the semantic routine, the parser resumes operation. Thus, the following are examples of semantic action calls:

<HEAD> ::= #BEGIN @S 3 <DECLARATION PART> #; @S ENDECPART

<TYPE LIST> ::= <TYPE LIST> #, <*I> @S 7 | <*I> @S 7

The constructs above are all valid in TWINKLE. The symbol # precedes every terminal and the construct <*I> is the meta-terminal <identifier>. Similarly

<*N> is any number and <*S> is any string. Note also that, in the definition of semantic action calls, <identifier> is taken in a more general sense to include a beginning digit. Thus numbers are also legal names for semantic actions. Only BNF-like TWINKLE constructs are used for exemplifying the syntax-semantics links. The addition of semantic calls to the other TWINKLE constructs is similar and is discussed in Chapter 6. For a complete description of the syntax of the semantic action calls, see Mercer [3].

The first example above means that, when recognizing a <HEAD>, immediately after placing #BEGIN in MSTACK, the semantic action named "3" must be executed. Then, after the recognition of a <DECLARATION PART> and a ";" the semantic action named "ENDECPART" should be executed. Note that when an action call is at the end of a production, it is executed immediately before the reduction takes place. In the second example, a <TYPE LIST> is recursively defined as identifiers separated by ",". After each identifier is placed in MSTACK, the semantic action named "7" is to be executed.

It should be recalled that both BIGTAB and MSTACK are in the scope of the semantic routines and have fields labelled SEM which are reserved for semantic information. Thus one of the things often performed by a semantic action is to insert semantic information in a MSTACK or BIGTAB SEM field thus associating "semantics" with the syntax that was placed there by the parser. All the fields of MSTACK or BIGTAB can also be interrogated by a semantic action to retrieve information left there by previous executions of semantic actions or by the parser. Semantic routines are also used for many other things such as building identifier tables and generating object or intermediate language code.

It is vital that the user know what the situation of MSTACK will be when a semantic action is called. In the first example, for instance, when the routine ENDECPART is called, it is known that the positions TOP, TOP-1 and TOP-2 of MSTACK must contain, respectively, the terminal ";", the terminal or meta terminal that began the declaration part, and the terminal #BEGIN. Note also, tha

upon restarting of the parse, a reduction will occur thus destroying TOP and TOP-1 and leaving the previous TOP-2 as the new top of MSTACK. Therefore any useful semantic information that might be stored in the SEM field of TOP or TOP-1 must be saved somewhere else by ENDECPART or it will be lost.

2.3.2.2 Semantic Tests

Consider the two following BNF productions that are present in the syntactic description of DEMALGOL I:

<ARITHMETIC PRIMARY> ::= <*I> @S 17

<BOOLEAN PRIMARY> ::= <*I> @S 19

The language thus defined is obviously ambiguous if one considers that an arithmetic assignment statement and a boolean assignment statement may both be of the form <*I> ← <*I> . Thus context cannot resolve the conflict and there is ambiguity, i.e., the parser cannot know, at that point, whether to call action "17" and make a reduction to an <ARITHMETIC PRIMARY> or to call "19" and make a reduction to a <BOOLEAN PRIMARY> . There is, however, one way to solve the problem: if the identifier has been declared of type INTEGER then the first production must apply; else, if it has been declared of type BOOLEAN the second production is to be chosen. This is, in essence, syntactic information since it would be possible to write a syntactic description of DEMALGOL I that would "remember" what an identifier had been declared. Such a description, however, would have to use Context Sensitive productions. Therefore, the difficulty arises from the fact that BNF is not powerful enough to describe all of DEMALGOL I (or all of any other nontrivial ALGOL-like language). The solution is to describe whatever subset of the syntax BNF is capable of and leave the rest to the semantics. Thus, semantic routines must keep tables of what type each identifier has been declared and they should be able to transmit this information to the parser at the appropriate time. This is done by

means of a special type of semantic action call called semantic test which is placed immediately following the symbol (terminal, meta-terminal or non-terminal) that would have caused the ambiguity. Syntactically, a semantic test is similar to a semantic call except that @T is used instead of @S. A semantic test routine may do anything that a regular semantic routine may do. It must, however, also set a fixed global boolean variable, called SEMANTICTEST, either true or false recording as the symbol in the stack is semantically matched or not, respectively. A semantic test could be described technically as a method for the parser to retrieve, from the semantics, information which is actually syntactic but which was left to the semantics due to the impossibility of describing context sensitive languages with BNF productions.

Thus, in the example given at the beginning of this section, one should add a semantic test to either one of the two productions. For example, the first production could be made.

<ARITHMETIC PRIMARY> ::= <*I> @T 10 @S 17

Semantic routine "10" must perform as one of its acts the following: check the table of declarations (that must have been previously built by other semantic actions) and set SEMANTICTEST to true if the identifier which is on top of MSTACK has been declared of type INTEGER, set it to false in case of a type BOOLEAN (DEMALGOL I has no REAL type). Note that a semantic test must immediately follow the term it modifies. Therefore, no semantic test may be preceded by a semantic action call. The use of semantic tests presents some difficulties which are described by Beals [2], page 33.

2.3.2.3 Other Links

It has been shown that the main system of syntax-semantics links in the system is provided by:

a) semantic action calls

b) semantic tests

c) the global boolean variable SEMANTICTEST

d) presence, in MSTACK and BIGTAB, of both syntactic and semantic fields allowing association of semantics to syntactic elements.

However, recalling the fact that all the scanner and parser procedures and variables are global to the semantic routines and thus are in their scope (see Figure 4), all these elements could be considered secondary syntax-semantics links since they can be checked and/or modified by a semantic routine. Three of these secondary links are used very often. Firstly, the line printer file, in which the parser messages are written, may also be used by a semantic routine to write some information (probably debugging aids). Thus syntactic and semantic messages will appear printed together, in the correct sequence. Secondly, the integer variable NUMERRS which the parser increments each time it finds an error, may also be incremented by semantic routines each time a semantic error is detected (for example, an identifier used in an expression has never been declared). Thus, NUMERRS = 0 at the end of a compilation will mean completely successful compilation rather than simply no syntactic errors. Thirdly, the scanner control variables (see Table 3) may be modified by a semantic routine thus providing the ability to change the scanner behavior when recognizing particular types of constructs (for example, when parsing a FORMAT declaration I5 should be recognized character by character instead of as an identifier).

It should be noted that all the discussion of syntax-semantics links was for pass 1 only. For pass i (i>1), the same principles apply except that (obviously), semantic tests, BIGTAB and scanner control variables do not exist any more.

2.3.3 The Floyd Productions Generator

The main task of the FP Generator is to transform the BNF tabular description of the syntax (received from the TWINKLE translator in PROTAB of TABLESF) to a set of Floyd Productions (FP's) which are the basis of a deterministic parsing algorithm. This program and the transformation algorithm BNF → FPL have been described by Beals [2]. TABLESF is also used as output by the FP Generator, however its main output, i.e. the set of FP's, is written on B-5500 disk as a separate file called FLOYDP. The reason for this is that the file is large and it is needed only between the FP Generator and the PT Generator. Therefore it would be a waste of space to keep it in TABLESF at all times as is done with most of the other tables.

2.3.4 The Parser Tables Generator

The main task of the PT Generator is to transform the FP's received from the FP Generator in the file FLOYDP into a very compact set of "parser instructions" and other tables that can actually be used by the parser. TABLESF is also used as both input and output. The table of "parser instructions" is FPTAB of TABLESF and it contains triples of words. Each triple contains all the information needed for implementing a Floyd Production such as: stack tests to be performed, semantic tests to be performed, lookahead symbols to check, which semantic routine(s) to call if the tests succeed, stack reductions to be made, and which FP to apply next.

2.3.5 Interpretive and Executable Parts - The ALGOL Code Generator.

In order to explain the function of the ALGOL Code Generators, the concept of interpretive code must be established and compared with the concept of non-interpretive (or executable) code. Non-interpretive or executable code is taken here in its usual meaning: a piece of code that can be directly executed by a machine, i.e., machine code. Interpretive code is understood

as a piece of code to be "executed" by an interpreter, i.e., a program that reads the interpretive code and "obeys" it. Thus the interpreter has actually the role of a simulator that simulates in the real machine an imaginary machine that accepts the interpretive code. Interpretive systems are frequently called "table-driven". Their advantage is the use of a more compact program; the interpreter, usually small, is the only program always in core and the instructions for the interpreter are read from the tables as needed. On the other hand, an executable system allows faster execution (since the simulation is avoided) usually at the expenses of a larger program since the previous table has been transformed into a program. Thus this seems to be another application of the classical speed-space trade-off, except that in the B-5500 system this trade-off ultimately leads to a choice of slow execution and little overlay versus fast execution and much overlay. Therefore, in the B-5500, an executable system may end up to be slower (in total time) than an interpretive system since, for very large tables, the resulting executable program is very large and the shorter execution time is more than compensated by the larger overlay time.

At the end of the syntactic preprocessor, one must choose between an interpretive system and an executable system. Three different pieces of the final compiler may be either interpretive or executable: the FP package, the LK procedure and the FILLTAB procedure (those pieces are marked by a dotted underline in Figure 4). Any combination of interpretive and executable for these three pieces is allowed and can be requested by using TWINKLE control options. If one of the pieces is interpretive, then an interpreter for it is taken out of the TWS Symbolic Library and inserted in its place in the compiler. At compile time, this interpreter will use TABLESF as its source of code. This was indicated in Figure 10 by dotted lines. If a piece is executable, then the 4th program of the syntactic preprocessor must be run. This program, the ALGOL Code Generator, takes the code for the piece out of

TABLESF and translates it into ALGOL Code which is then inserted in the proper position in the compiler. That part then does not need TABLESF at compile time. This possibility was indicated in Figure 10 with dashed lines. The two extreme cases are:

a) All 3 pieces interpretive. In this case, the ALGOL Code Generator does not have to be run. The final compiler is language-dependent only due to the semantic routines. The syntactic analyser is general purpose and will receive at run time (and for each run) the information about the syntax of L from TABLESF. This compiler is faster to generate (since one of the programs is not used) and is probably the best to use for the debugging stage of implementation of a new language with the system.

b) All 3 pieces executable. In this case the final compiler contains in itself all the information about the language (i.e., both syntax and semantics) and TABLESF is not needed at compile time. For small languages, this will probably be the choice for the final compiler since it is likely to be the fastest. For large languages only timing tests can indicate the best combination of executable and interpretive pieces.

2.3.6 The ISL Translator

The inputs for the ISL translator are the semantic description of the language in ISL (either input directly or given to the TWINKLE translator and passed to the ISL translator as part of the file ACTIONS) and the file ACTIONS providing information about the code number TWINKLE associated with each semantic action name found in the syntax. The ISL translator must then associate the same code number to the definitions of those actions in order to match a semantic action definition with its call. The main task of the ISL translator is to translate each piece of the semantic description

(one for each pass and, if the number of passes is not 1, also one for HEAD) into ALGOL and insert it into the appropriate position in the compiler. Since ISL is basically an extension of ALGOL, the ISL translator must detect the additional (i.e., non-ALGOL) constructs and translate them into ALGOL preserving, unmodified, what was already in ALGOL. It must also build the procedure EXEC(n) described in Section 2.2.1.3 which must be available for the parser to call when it has to obey a semantic action call.

This completes the general description of the complete TWS System. The rest of this paper will be devoted to a detailed discussion of ISL.

3. DESIGN AND IMPLEMENTATION

This chapter contains a discussion of the design of ISL and the approaches adopted in its implementation.

3.1 Design Considerations

3.1.1 Critique of FSL

The Formal Semantic Language (FSL) introduced by Feldman [10] and discussed by Northcote [11] was a major advance in the art of implementing TWS's. Its main advantage is the facilities provided for declaring and manipulating tables, stacks and other data structures not usually available in general purpose programming languages. This simplified the writing of a brute-force semantic description of a language and the resulting program is, to a certain extent, clear and self-documenting. However, if one considers the goals and characteristics of the ILLIAC IV TWS, the inclusion of FSL in this system would present problems in the following four different areas.

1) Although FSL embodies several ALGOL features, its notation is, in several cases, considerably different than the ALGOL one. In fact, certain representations are downright cumbersome. In a table operand, for example, the user must submit a string of commas with a "$" in the proper place to indicate which column of the table is to be referenced. This makes human adaptation from ALGOL to FSL a non-trivial effort.

2) It seems that FSL, probably unintentionally, embodies some particularizations that make it specially powerful only for implementing one-pass compilers for ALGOL-like languages to be run in a conventional (sequential) computer Despite the fact that a certain amount of particularization is probably unavoidable (and may be even desirable) in order to generate good code, the restrictions

above could not be tolerated in a system whose basic feature is the production of n-pass compilers and the main application of which has been to implement languages for a highly parallel machine.

3) With regard to (2) above, the run-time features (RTABLE, RSTACK, etc.) and the use of the CODE brackets to manipulate them seem to be a particularly weak area. Besides being complicated to use, the implementation of these facilities is extremely machine-dependent. Machine independence was to be obtained by having "the machine dependent aspects of primitives embedded in the compiler kernels". This would not, however, fit in the ILLIAC IV TWS approach of leaving machine dependence to be dealt with, at pass 2, by user implemented IL operators.

4) Being a compiler language, FSL needs a complicated translator. This either limits the use of FSL to the few machines for which such a translator is available or obliges the user to undertake the major project of writing one himself.

3.1.2 The ISL Approach

Since the facilities for declaring and manipulating tables, stacks, etc. were considered to be one of the main advantages of FSL, they were retained in ISL. The problems identified in FSL were dealt with in ISL using the following approaches:

1) ISL notation was made as ALGOL-like as possible. In fact, most ISL constructs very closely resemble ALGOL procedure calls. Besides, the simple fact that ISL is an extension of ALGOL makes ALGOL available as part of ISL thus making the implementation of anything already present in ALGOL unnecessary. Therefore ALGOL programmers (and, in particular, BeA programmers) find ISL a very "natural" extension of ALGOL, easy to learn and use.

2) Several FSL features which were deemed to be too directly concerned with the implementation of one-pass compilers were not defined in ISL since they

are troublesome to implement and probably would not be very useful in the system. This includes FSL's FLAD, CHAIN and ASSIGN statements. When needed, the equivalent of these constructs can be easily simulated using regular ISL or BeA features.

3) No facilities are provided in ISL to generate machine code or to declare and manipulate run-time stacks and tables. It was felt that the implementation of these features is too machine dependent and, according to the generalized IL philosophy, should be left to user defined IL operators. Normally machine code, for example, is generated with BeA WRITE statements in the semantic routines of the last pass. This also allows great freedom in the FORMAT of the output so that when assembly language is to be the final result, it can be written in the format expected by a particular assembler. While the machine language output is provided by the user, intermediate language output and input statements are available in ISL. This is made possible by the fixed, simple syntax of the IL. Intermediate language operators and operands can be emitted using the ISL CODE statement. IL input is automatically provided by the IL Recognizer. However, the user can also cause the IL string to be read by using the procedures described in section 2.2.2.

4) The approach of defining ISL as an extension to ALGOL leads to two great advantages. Firstly, all the power of BeA is given to the ISL user including such valuable features as multi-dimensional arrays and recursive procedures. The power of these constructs guarantees that the most tricky semantic actions may be described in ISL. Secondly, ISL can be easily implemented with a two-pass compiler in which the first pass detects the ISL (i.e., non-ALGOL) constructs and translates them into ALGOL preserving, unmodified, the ALGOL constructs. A pure BeA program is then obtained which is translated, in the second pass, to machine code by the BeA compiler. Thus the implementation of ISL in a new machine involves only the relatively simple conversion of the first pass (henceforth called the ISL translator) from BeA to the local version of ALGOL

since an ALGOL compiler is usually available. Obviously this approach will yield good code only as far as the ISL constructs can be efficiently defined in terms of ALGOL. This was possible in the system's ISL translator due to the availability, in BeA, of bit manipulation facilities. Since these facilities are not likely to be encountered in other ALGOL versions, the implementation of ISL for a non-Burroughs machine may not be as simple as outlined above.

3.2 Implementation

Two different ISL translators were written for the system: a brute-force ISL translator and a TWS ISL translator. Only the brute-force translator will be described in some detail, but speed comparisons will be given for the two versions.

3.2.1 The Brute-Force ISL Translator

The brute-force ISL translator was manually written in BeA. It is a program consisting of approximately 3000 source cards and translates a "typical" ISL program into BeA at a rate of about 2000 cards per minute. The test program used was an actual 2000 card long semantic description of a language and the typical density of ISL constructs is of the order of one construct per two cards.

Since brute-force semantic descriptions of complex programming languages can be extremely large, it was vital that the ISL translator have a good speed. In order to attain this goal, the syntax of ISL was carefully formulated to allow efficient parsing by the ISL translator, i.e., ISL constructs must be quickly recognized and isolated with a minimum parsing of the ALGOL constructs interspersed among them. This was done by defining that all ISL constructs be characterized by an identifier immediately preceded by the character "$" (dollar sign). Moreover, the appearance of the "$" in column one of a card was forbidden. A free "$" (i.e., not embedded in a string) cannot occur in BeA except in column 1 (and then it indicates a BeA control card). When an ISL construct has

parameters they must appear, enclosed in parentheses, after the $ <ISL construct identifier> that mark it.

Figure 11 presents the basic block structure of the brute-force ISL translator. The main part is the "basic parsing algorithm" which is treated in the next section. Then there is one block of code for each ISL construct. The basic parsing algorithm gives control to the block of code that handles a specific ISL construct when it has found that construct. Moreover, the string of arguments (if any) for the construct is known to be ISL-free (i.e., nesting has been taken care of) and is placed in an array called ARGUMENT. The locations STARTPAR and ENDPAR are also set by the parsing algorithm to contain the locations in ARGUMENT at which the string of parameters start and end. With this information, the block of code that handles the construct can translate it, put the resulting BeA code in the array RESULT and call a procedure named RETURN that picks the result and gives control back to the parsing algorithm. These blocks of code are referred to as IMPLXXX where XXX stands for the name of the ISL construct the block handles. The IMPLXXX blocks are relatively straightforward and their implementation will not be described except for some comments, in Chapter 4, about the type of BeA code generated by the different ISL constructs. The basic parsing algorithm, however, deserves further discussion.

3.2.1.1 The Basic Parsing Algorithm

The effect of the parsing algorithm was described in the preceding section. Its main task is to take care of nesting of ISL constructs so that when several ISL constructs appear nested the parsing algorithm first calls IMPLXXX for the innermost construct. The result of this construct is then inserted in place and scanned for further ISL constructs (thus an ISL construct may generate other ISL constructs in a local multi-pass translation). When there are none, IMPLXXX is called for the second construct (from the inside) which is now known not to contain ISL in its string of parameters and the

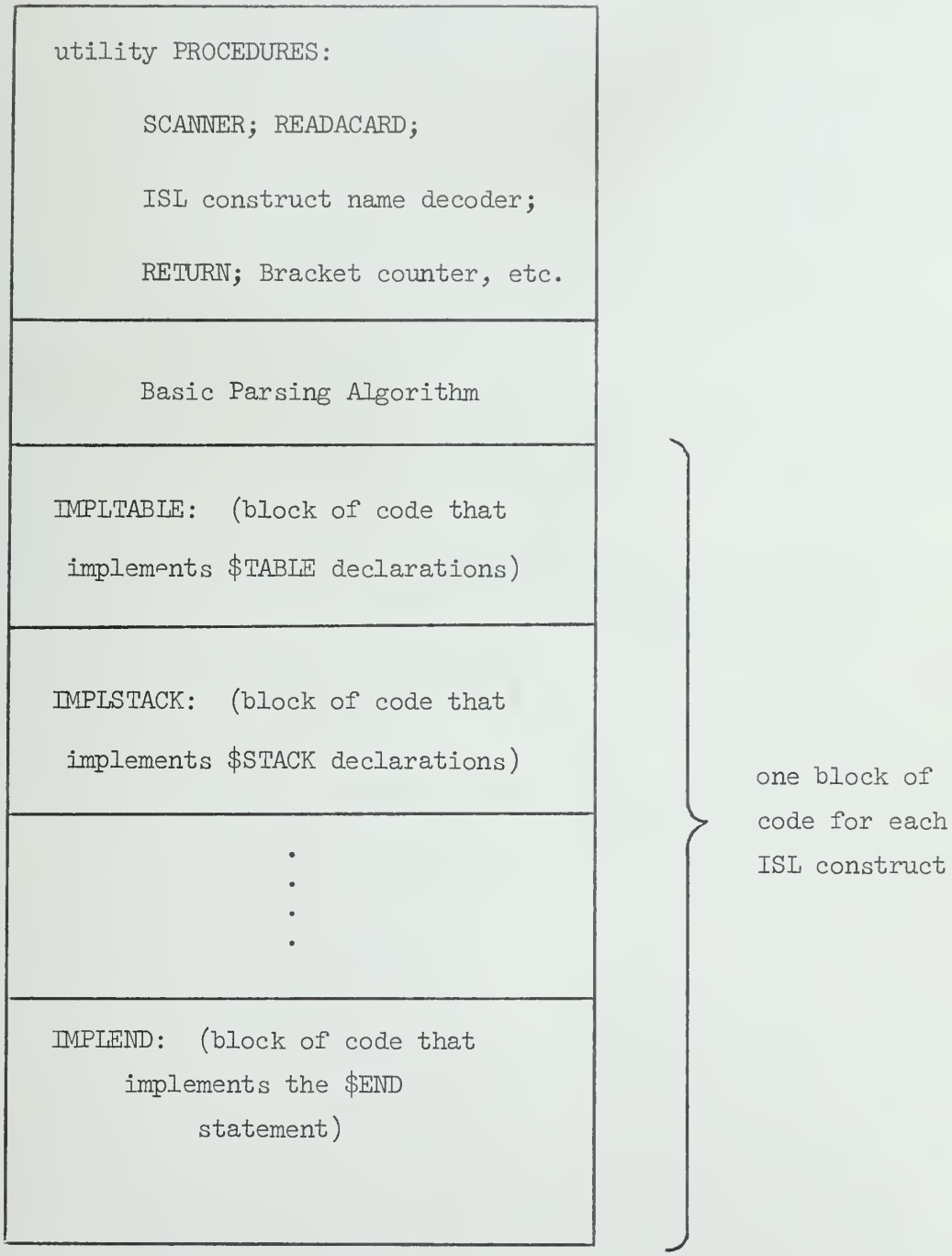


Figure 11. Block diagram of the brute-force ISL translator.

procedure continues until the outermost construct is translated. The basic element in the analysis of nesting is a stack called NESTACK. Its two main fields are CONSTRUCT which stores the name of the construct and STARTPAR which points to the location in ARGUMENT at which the string of parameters for this construct begins. The functioning of the parsing algorithm can best be understood by studying Figure 12 which presents a flow diagram for the brute-force ISL translator detailing utilization of the parser. Note that the scanner scans elements from INPUT which may be either a card image or the array ARGUMENT. Note also that the top of NESTACK contains CONSTRUCT and STARTPAR for the construct presently being translated while the positions further down contain that information for successive outer levels of nesting.

3.2.1.2 The Parsing Algorithm as Applied to the Implementation of a Class of ALGOL Extensions

In Figure 11, one can notice that the only parts which depend on particular ISL constructs are the IMPLXXX blocks and the ISL construct name decoder. The other procedures and, particularly, the basic parsing algorithm could be used to recognize any extension of ALGOL with non-ALGOL constructs of the type:

$<construct identifier> (<string of parameters>)

All the user would have to do is to provide an IMPLXXX block for each construct and to insert, in the construct name decoder, the names (identifiers) associated with each construct. At first, it seems that these extensions of ALGOL might as well be implemented with a standard deck of ALGOL procedures which would be inserted at the beginning of each program, each procedure defining (or implementing) one construct. However, the extensions that can be obtained with this simple method are very trivial due to the restrictions imposed by the ALGOL rules for use of procedures. Much more sophisticated extensions can be implemented using the kernel of the ISL translator as described above. In fact, this

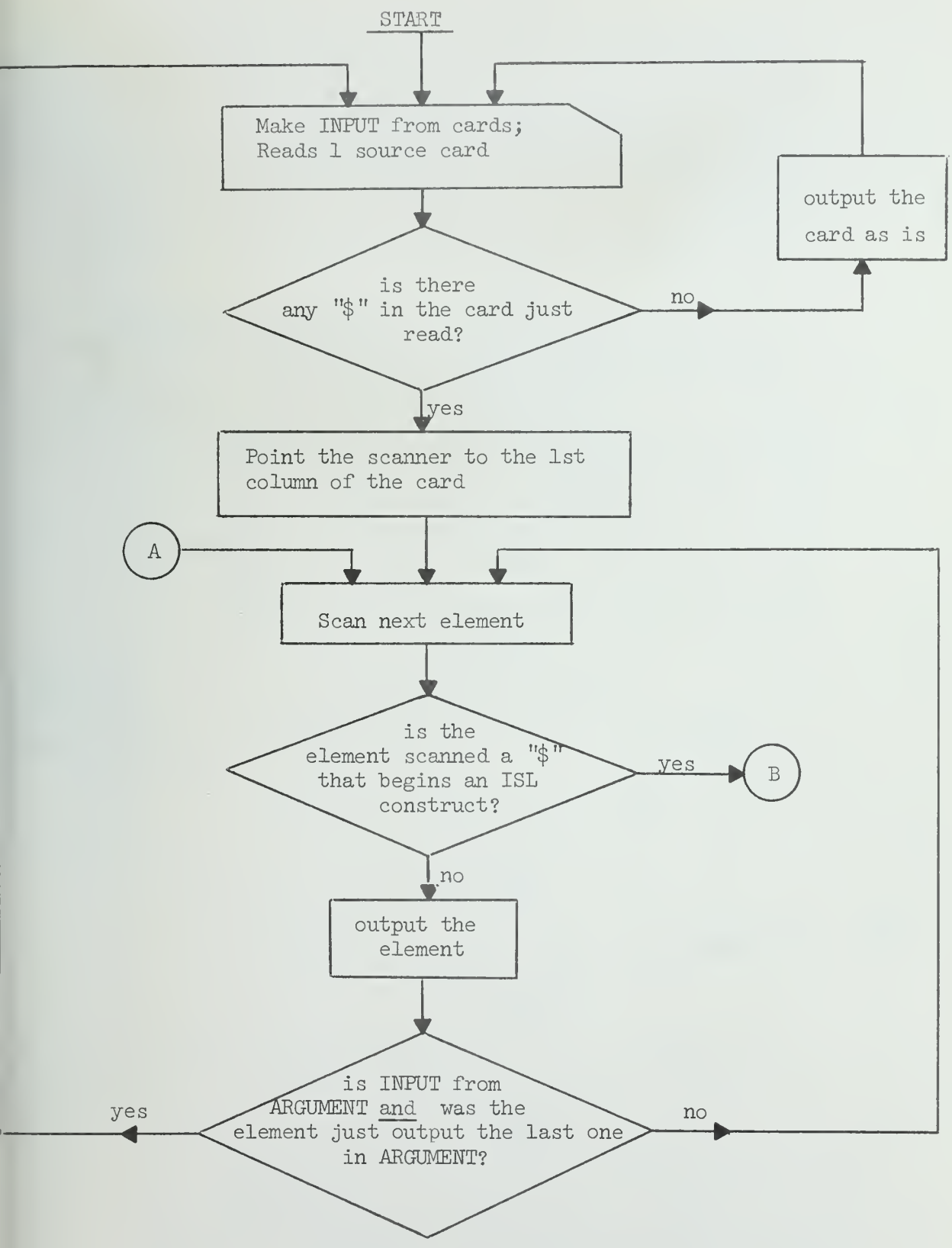


Figure 12. Flow diagram of the brute-force ISL translator.

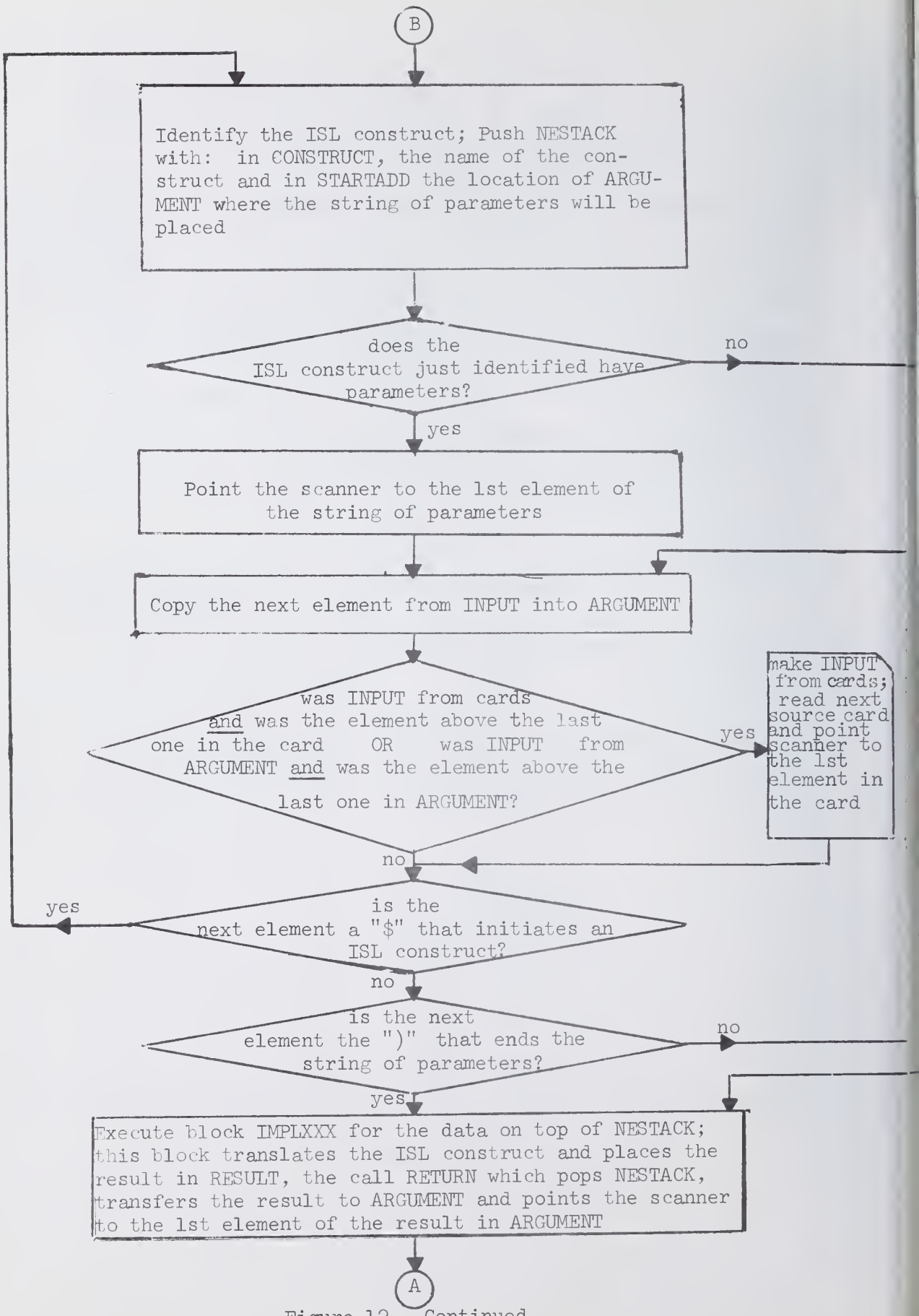


Figure 12. Continued.

method could be considered a very primitive TWS in which the syntax is almost fixed and the IMPLXXX blocks are the semantic routines. A number of set declarations and set manipulation statements were defined in a SET-ALGOL extension to test the application of the kernel of the ISL translator in its implementation. The method worked well and SET-ALGOL was partially implemented using such a kernel.

3.2.2 The TWS-ISL Translator

Once the brute-force ISL translator was completed and integrated into the system the TWS itself could be used to implement ISL. This was actually done and the resulting one-pass translator has been named the TWS-ISL translator. Note that for this application the semantics of ISL were described in ISL. The advantages of the TWS-ISL translator over the brute-force one are basically two. First, the TWS-ISL translator is "easily" modifiable by anyone familiar with the TWS system. This is a very important feature, as it facilitates future extensions of ISL. Secondly, TWS-ISL is a more sophisticated version of ISL than brute-force-ISL.

The main disadvantage of the TWS-ISL translator, when compared with the brute-FORCE one, is its speed: approximately 500 cards per minute against 2000 for the brute-force. This is, in fact, a remarkable performance for the TWS-ISL translator since it does not have the initial scanning to deal immediately with non-ISL cards which is largely responsible for the speed of the carefully hand coded brute-force translator. It is hoped that the introduction of this optimization and improvements being completed in several areas of the system will boost the speed of the TWS-ISL translator and make it comparable with the brute-force one. This will then be discarded and the system will be "closed", i.e., both the syntactic and semantic metalanguages will use translators implemented with the aid of the system itself.

4. A DESCRIPTION OF ISL

This chapter presents a description of ISL. Semantics is described in English and syntax definitions are given using BNF productions in an informal way or via English descriptions. The formal BNF syntax for ISL is presented in Appendix A, each production being numbered. Examples for each construct will be taken, whenever possible, from the semantic description of DEMALGOL I given in Appendix D and discussed in Chapter 6.

4.1 General Structure of ISL

Pure ISL constructs (i.e., non-ALGOL) may be classified into three general categories: ISL declarations, ISL statements and ISL operands. ISL declarations are used similarly to ALGOL declarations and serve mainly to declare the ISL data structures: stacks, tables and cells. Each structure must be declared before it can be used in a statement or operand. ALGOL and pure ISL declarations may be freely intermixed in the declaration part of a block while ALGOL and pure ISL statements and operands may be used in any order in the compound tail of each block. As in ALGOL, no declaration is allowed among the statements (except if a statement is a block) and no statement is allowed among the declarations. It is easy to see the reasons for these rules if one considers that ISL statements and declarations are translated into ALGOL statements and declarations and a valid ALGOL program must result. The ALGOL rules about use of the same identifier in different blocks and scope of each declaration also apply to ISL.

Every ISL construct is characterized by the character "$" immediately followed by the name of the construct which is an identifier with six or fewer characters. When the construct has parameters, they must appear, enclosed in parentheses, following the $ <ISL construct identifier> . Moreover, the "$" may never be in the first column of a card.

Every ISL program must start with the construct $BEGIN and end with $END. These replace the initial BEGIN and final END of an ALGOL program and serve as markers for the ISL translator to begin and to quit translating. Note that $BEGIN appears only at the beginning of the program and $END only at the end.

An ISL program is the semantic description of one pass of a compiler for a given language. Thus, the complete semantic description of a language L to be implemented with an n-pass compiler requires n ISL programs, one for each pass (additionally, if n > 1, a HEAD containing only a <list of declarations> as defined in Appendix A, production 10). These n programs can be fed sequentially in only one file to the ISL translator or the TWINKLE translator as described in Chapter 5.

4.1.1 The Action Label

It has already been mentioned in Chapter 2 that the basic semantic unit is the semantic action or semantic routine which is called by semantic action calls and semantic tests in the syntax. Therefore, the compound tail of the main block of an ISL program is divided into pieces which correspond to the semantic actions. Each of these pieces is composed of one or more statements and is identified by an ISL construct called action label which is used just like a label in ALGOL and has the following syntax:

<action label> ::= $ACTION (<generalized identifier>)

where a generalized identifier is an identifier in which not only letters but also digits are allowed as the first character. Normally, a generalized identifier appearing in an action label must have appeared somewhere in the syntax as the name of a semantic action in a semantic action call or semantic test. In general, action labels do not have to be declared since their "declaration" is provided to the ISL translator by the TWINKLE translator via the file ACTIONS.

An action label must precede the first statement of each semantic action and every statement in an ISL program must belong to a specific semantic action (except if the statement is in a procedure body in the outermost block). The end of a semantic action is automatically indicated either by the next action label or by the $END. Note that the specifications above result in the fact that the declaration part of the outermost block of an ISL program must be followed by an action label. Thus the appearance of the first action label marks to the translator the end of the outermost declaration part. This is important since the translator must know that point and can find it without an extensive parsing of the declarations. Figure 13 presents the block structure of a typical ISL program.

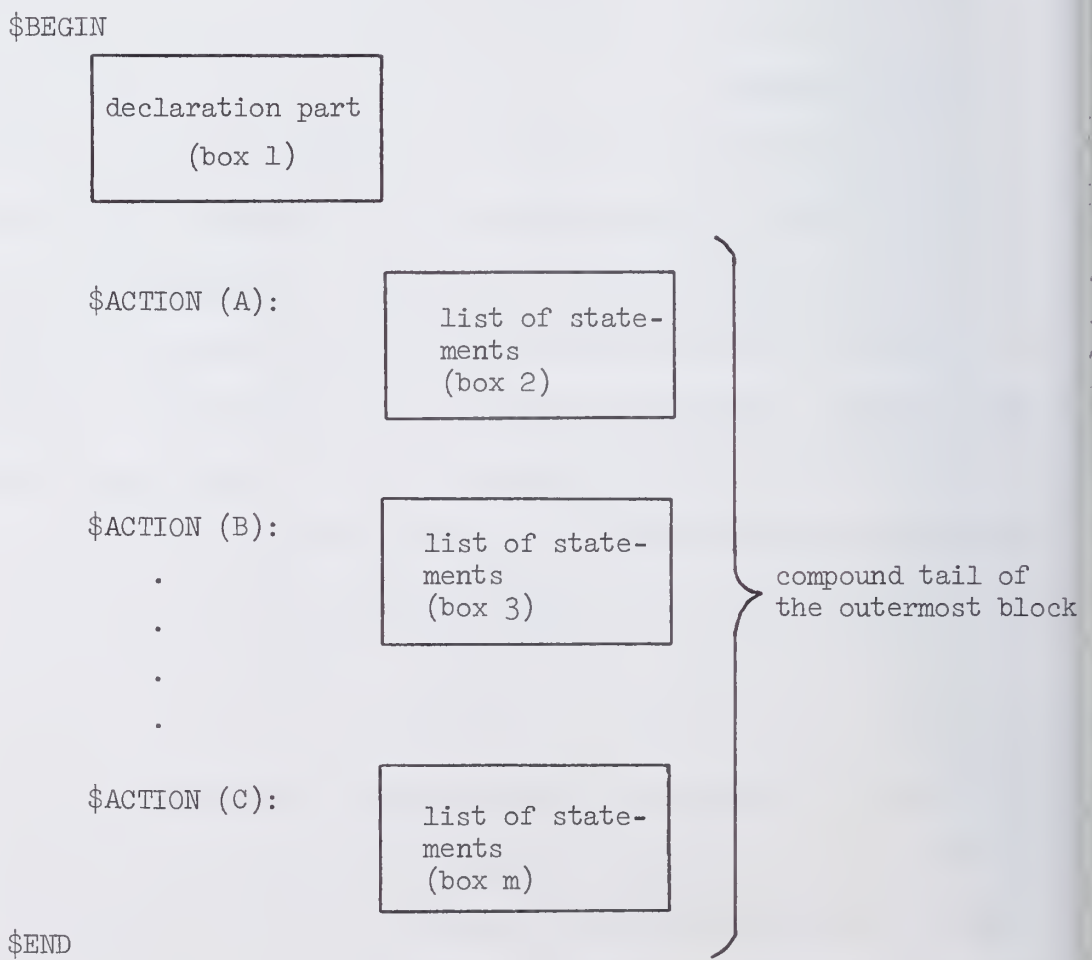


Figure 13. Block structure of a typical ISL program.

4.1.2 Semantic Action Zero

The name "0" (character zero) has a special meaning in a semantic action: it can appear in an action label in an ISL program but it cannot appear in a semantic action call in the syntax. In fact, this action is automatically called by the parser before the parsing starts at each execution of the compiler (see Figure 4 for an indication of where action 0 is called). For pass i (i > 1), the IL recognizer calls action 0 before starting reading the IL string. Therefore, the user may use action 0 to perform all initializations needed in the semantic variables, tables, stacks, etc. The ISL translator also generates some variables that need initialization; for each table or stack declared in a block, a pointer is created. These pointers must be initialized to zero at the beginning of the compound tail of the block at which they were created. For the outermost block, such pointer initialization is performed in action 0. Thus, if the user declared an action 0, the ISL translator introduces at its beginning the pointer initializations. If the user did not use action 0, the ISL translator creates one as the last non-empty action and in this case all it contains is the pointer initializations.

4.1.3 The Structure of a Translated ISL Program

Some considerations about the code generated by the ISL translator are in order since they help in understanding a few of the ISL constructs. Figure 14 presents the block structure of the code generated by the ISL translator for the ISL program in Figure 13 and file ACTIONS as in the upper right corner of Figure 14. First of all, it should be noticed that the output code is not a complete ALGØL program; it consists of declarations and a few executable statements at the end. Thus, in Figure 4 the output of the ISL translator is not only the semantics package but also the language assignments and the call to PARSE and PEND; in Figure 8, the output of the ISL translator is the pass i semantics package and the call to the IL recognizer. Therefore, the executable

file ACTIONS

action name	action code #
B	1
XXX	2
A	3
XXX	4
C	5
XXX	6
.	.
.	.
.	.

```
standard declarations

declaration part
(box 1 translated to ALGOL)

PROCEDURE EXEC(N); VALUE N; INTEGER N;
   BEGIN
      LABEL ACTIONA, ACTIONB, . . . , ACTIONC, ISLEND;
      SWITCH ISLSWITCH:= ACTIONO, ACTIONB, ACTIONXXX, ACTIONA, ACTIONXXX, ACTIONC,
                           0       1        2         3        4         5
                         ACTIONXXX, . . . ;
                            6
      GO TO ISLSWITCH (N + 1);
      ACTIONA:    list of statements
                  (box 2 translated to ALGOL)
            GO TO ISLEND;
      ACTIONB:    list of statements
                  (box 3 translated to ALGOL)
            GO TO ISLEND;
         .
         .
         .
      ACTIONC:    list of statements
                  (box m translated to ALGOL)
            GO TO ISLEND;
      ACTIONO:    pointer initializations
            GO TO ISLEND;
      ACTIONXXX: ACTIONXXX: . . . ACTIONXXX: ISLEND:
   END OF PROCEDURE EXEC;

executable part
```

Figure 14. Block structure of a translated ISL program.

part at the bottom of Figure 14 consists mainly of calls to the IL recognizer for pass i (i > 1) and of calls to PARSE and PEND for pass 1. The declarations can be divided into three parts:

a) a group of standard declarations containing variables, procedures and definitions used throughout every translated ISL code;
b) the user's declaration part which is simply box 1 of Figure 13 with the ISL declarations translated into ALGOL declarations;
c) procedure EXEC(N).

Procedure EXEC is the main procedure in the semantics package and the parser expects EXEC to behave in the following way: when EXEC(n) is called, the semantic action whose code number is n must be executed, and no other statement should be executed. Figure 14 shows clearly how this is accomplished with the aid of the switch ISLSWITCH. One ALGOL label is generated for each action name by concatenating the ACTION name to the identifier ACTION. Then each of these labels is placed in switch ISLSWITCH, in the i-th position if the code number of the action is i. All over Figure 14 XXX stands for other action names. In this example it was assumed that the user did not use action 0 in the semantics. Therefore, the ISL translator introduced it, containing only pointer initializations, after the last semantic action. Finally, after action 0 are placed the labels corresponding to action names that occurred in the syntax but did not occur in the semantics so they are treated as empty actions. Note that the ISL translator has only one pass, so ISLSWITCH is generated based only on the information present in the file ACTIONS since at that point the action names that will appear in the semantics are not yet known.

4.2 ISL Declarations

The first category of ISL constructs, ISL declarations, can be loosely divided into three groups:

a) data structure declarations;

b) control declarations;

c) define declarations.

Each group will now be considered.

4.2.1 Data Structure Declarations

These declarations enable the user to establish three types of data structures useful in the description of semantics: tables, stacks and cells.

4.2.1.1 Table Declarations

The syntax for table declarations is as follows:

<table declaration> ::= $TABLE (<list of table specifications>)

<list of table specifications> ::= <table specification> |
<list of table specifications> ;
<table specification>

<table specification> ::= <table identifier> , <number of rows> ,
<list of column specifications>

<list of column specifications> ::= <column specification>|
<list of column specifications>,
<column specification>

<column specification> ::= <column identifier> = <left bit of field> :
<bits in field>

A table is a structure containing rows and columns. The table is designated by a table identifier which can be any identifier not yet declared in the block. The number of rows must be an unsigned integer and each row is designated by an integer: the first row is #0, the second #1 and so on. Each row contains a certain number of bits (constant for all rows of the same table) grouped in fields called columns. Each column is designated by an identifier. No two columns of the same table can be designated by the same identifier. However there is no restriction on the use of the same column identifiers in different tables, even when they are declared in the same block. The field associated with each column identifier is defined by a pair of unsigned integers separated by ":" . The first integer indicates the number of the first bit in the field and the second integer defines how many bits the field contains. Bits are numbered from left to right, starting with 0. Thus the column specification:

LEVEL = 32 : 4

associates with the column LEVEL the 4-bit field starting at bit # 32. Each table declaration automatically generates a declaration of an integer which is reserved as a pointer to the rows of that table. The identifier of the pointer of a table is formed by concatenating the identifier PT and the table identifier. Thus a table called IDTAB will have as pointer the identifier PTIDTAB. The pointer of a table may be used or set by the user and is also automatically used and/or updated by certain ISL constructs. Moreover, the pointer is automatically initialized to zero as explained in section 4.1.2.

Implementation and restrictions: The actual implementation of ISL in the B-5500 machine places some restrictions on the column specifications. Tables are implemented using arrays with the same identifier used for the table. The arrays can be one, two or three-dimensional. Each row contains (n DIV 48) +1 words where n is the bit number of the rightmost bit used in any column. Since the first bit (#0) of each word in the B-5500 is not accessible to the user,

bits whose number i is such that (i MOD 48) = 0 may not be used in any column. Tables which use only one word per row are more efficient to manipulate than tables with multi-word rows. Also tables with less than 1023 rows are more efficient to manipulate than tables with more than 1023 rows since these are simulated with a number of blocks of 512 rows each. The maximum value for the number of rows is 1023 × 512 = 523,776. The maximum bit number that can be included in a column is 1023 × 48 - 1 = 49,103. The column structure is implemented by emitting a number of BeA defines, one or two per column. Tables with single-word rows need one define per column and the identifier of the define is the concatenation of the table identifier with the column identifier. Besides these restrictions, tables with multi-word rows need one more define per column and the identifier is the same as that used for the first define with a "Z" added at the end. Thus a column named TYPE of a table called IDTAB generates the identifier IDTABTYPE if IDTAB has one word per row and the identifiers IDTABTYPE and IDTABTYPEZ if IDTAB has two or more words per row. One word about error detection in the ISL translator: ISL should be considered a 2-pass system, the first pass being the ISL translator and the second pass the BeA compiler. The ISL translator only detects some errors which are in the ISL constructs themselves but no ALGOL parse is done so many errors are found only at BeA compilation time.

Example of a table declaration:

```
$TABLE (IDTAB, 1022, PTTOBIGTAB =  1 : 13 , BACKPT   = 14 : 10 ,
                     PTTOABTAB  = 24 :  8 , LEVEL    = 32 :  4 ,
                     TYPE       = 36 :  2 , USED     = 38 :  1 ,
                     FOUND      = 39 :  1 , LABFIELD = 36 :  4 ,
                     PTTOLKLIST = 40 :  8 , REST     = 38 : 10 ;
              LKLIST, 256, PTTOABTAB  =  1 :  8 , LINK     =  9 :  8 ;
         BLOCKSTACK, 16, PTABTAB    =  1 :  8 , PTLKLIST =  9 :  8 )
```

This declaration defines three tables called IDTAB, LKLIST and BLOCKSTACK. Figure 15 presents the structure of the tables obtained with this declaration. Notice that column fields may overlap. For example, the column LABFIELD of IDTAB is in fact the union of the columns TYPE, USED and FOUND; the column REST of IDTAB is the union of the columns USED, FOUND and PTTOLKLIST. Notice also that, although more than one table may be declared inside the parenthesis of a $TABLE, the user is free to use as many $TABLE constructs as desired in the same declaration part.

4.2.1.2 Stack Declarations

The syntax for stack declarations is exactly similar to the syntax of table declarations except that no <list of column specifications> appears in a stack declaration. In fact, stacks are basically one-column tables, i.e., tables in which one word is allocated for each row. Therefore, there is no need to name this one column of a stack. Stacks also have a pointer automatically associated with them and the same formation rules as given for the table pointer apply.

In the B-5500 implementation, each position (i.e., row) of a stack is a 48-bit word. Like tables, stacks with less than 1023 positions are more efficient to manipulate for the reason already explained for tables. The maximum value for the number of rows in a stack is identical to that for tables.

Example of a stack declaration:

$STACK (IFSTACK, 32)

The structure of the stack thus declared is presented in Figure 16.

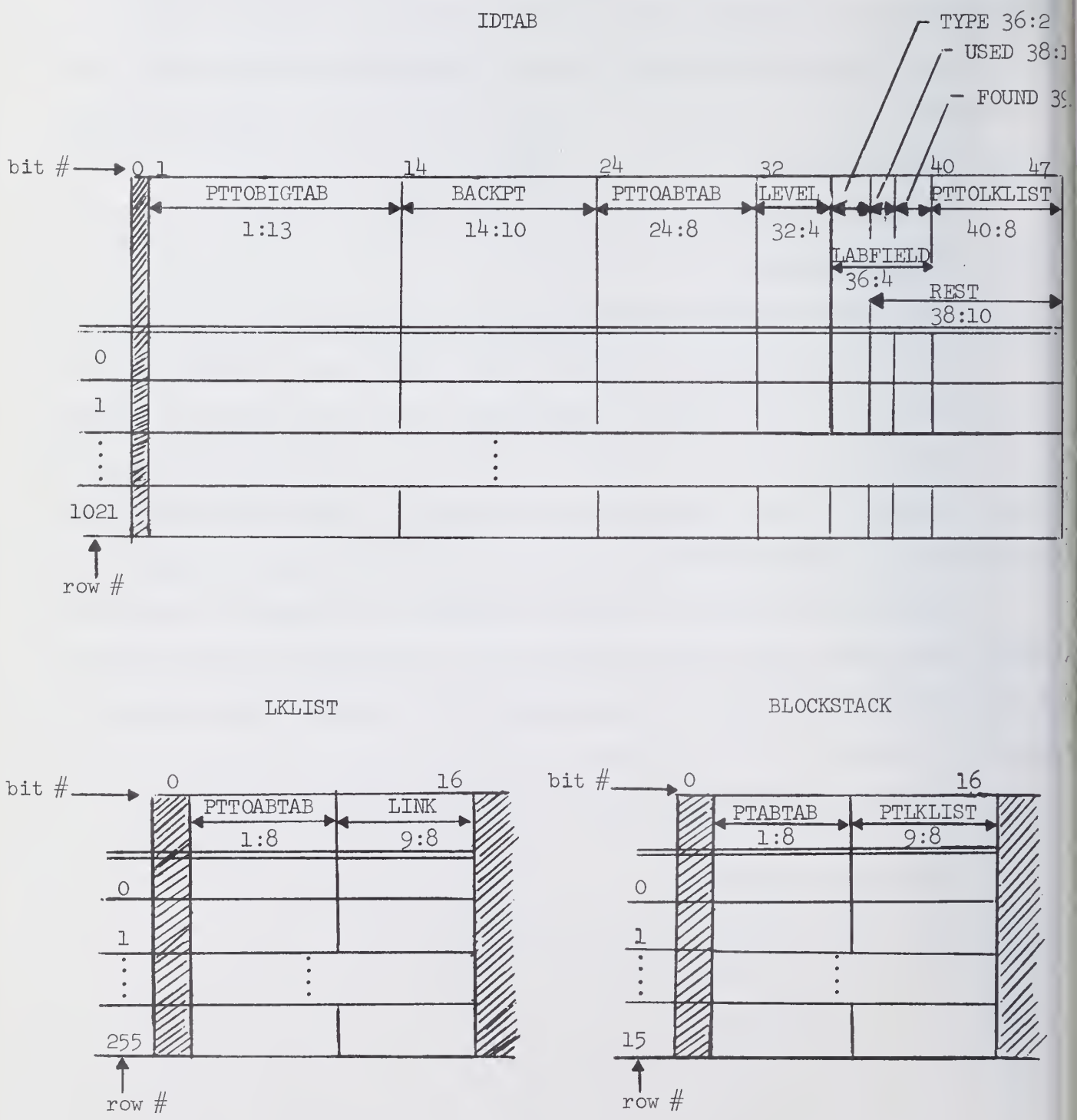


Figure 15. Structure of the tables IDTAB, LKLIST and BLOCKSTACK.

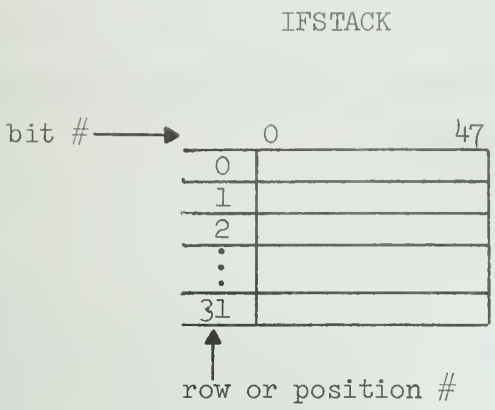


Figure 16. Structure of the stack IFSTACK.

4.2.1.3 Cell Declarations

The syntax for cell declarations is similar to that of table declarations, except that no <number of rows> appears in a cell declaration. In fact, cells are basically one-row tables. Therefore, cells do not have a pointer associated with them. Cells are useful to store patterns of information which will later be entered in a table with a column structure equal to the column structure of the cell. The same implementation techniques and restrictions mentioned for tables apply for cells: there may be one-word cells and multi-word cells and the former are more efficient to manipulate.

Example of a cell declaration:

```
$CELL (LKLISTCELL, PTTOABTAB = 1 : 8, LINK = 9 : 8)
```

This cell corresponds to one row of the table LKLIST, as presented in Figure 15.

4.2.1.4 The Predefined Data Structures and the Add Column Declaration

Certain tables are assumed to be predefined in an ISL program and the user does not have to (and may not) declare them. They are: MSTACK (for all passes) and BIGTAB (for pass 1 only). Therefore, the ISL translator behaves as if it received, at the beginning of each program, the following declaration (where n is the value of STACKSIZE):

```
$TABLE (MSTACK, n, LTYPE = 1 : 5, SEM = 6 : 12,
                   ENTRY = 18 : 12, TYPE = 30 : 6, ILENTRY = 36 : 12;
        BIGTAB, 8192, SEM = 1 : 15, WORDS = 16 : 3, CHARS = 19 : 3,
                   LEFTPT = 22 : 13, RIGHTPT = 35 : 13,
                   CHAR1 = 12 : 6, CHAR2 = 18 : 6, CHAR3 = 24 : 6,
                   CHAR4 = 30 : 6, CHAR5 = 36 : 6, CHAR6 = 42 : 6)
```

Figure 5 shows the structure of BIGTAB and Figure 7 the structure of MSTACK.

The add column declaration was created mainly to enable the user to modify the column structure of the predefined tables. This may be very convenient in order to divide the semantic field (which is predeclared simply as SEM) into several columns or to associate a name more mnemonic than SEM. Obviously the user cannot introduce these modifications in the ISL declaration of BIGTAB and MSTACK because such declarations do not appear physically. The add column declaration then may be used. Syntactically, the add column declaration is exactly similar to a cell declaration. The identifier, however, may be either a cell identifier or a table identifier provided it has already been used in the corresponding ISL declaration and the add column is in the scope of the "mother" declaration. An add column declaration does not create arrays or pointers: it just creates the BeA defines to add some new columns to the column structure already in effect for that table. Therefore, add column declarations are subjected to the restriction that the columns being added cannot require a greater number of words per row than the number already allocated to the table or cell

in the main declaration. Although primarily created because of the predefined tables, add column declarations may be used for any cell or table previously declared and any number of these declarations may modify the same main declaration.

Example of an add column declaration:

```
$ADDCOL (BIGTAB, PTTOIDTAB = 6 : 10, NUMBEROK = 5 : 1 ;
         MSTACK, PTTOIDTAB = 8 : 10, NUMBEROK = 7 : 1 )
```

This declaration declares two new columns in BIGTAB and MSTACK, both contained inside column SEM. Figure 17 shows the new structure of MSTACK and BIGTAB, after being modified by the declaration above.

4.2.2 Control Declarations

These declarations are mainly concerned with providing information for the ISL translator enabling it to generate correct ALGOL code.

4.2.2.1 Forward Action Declaration

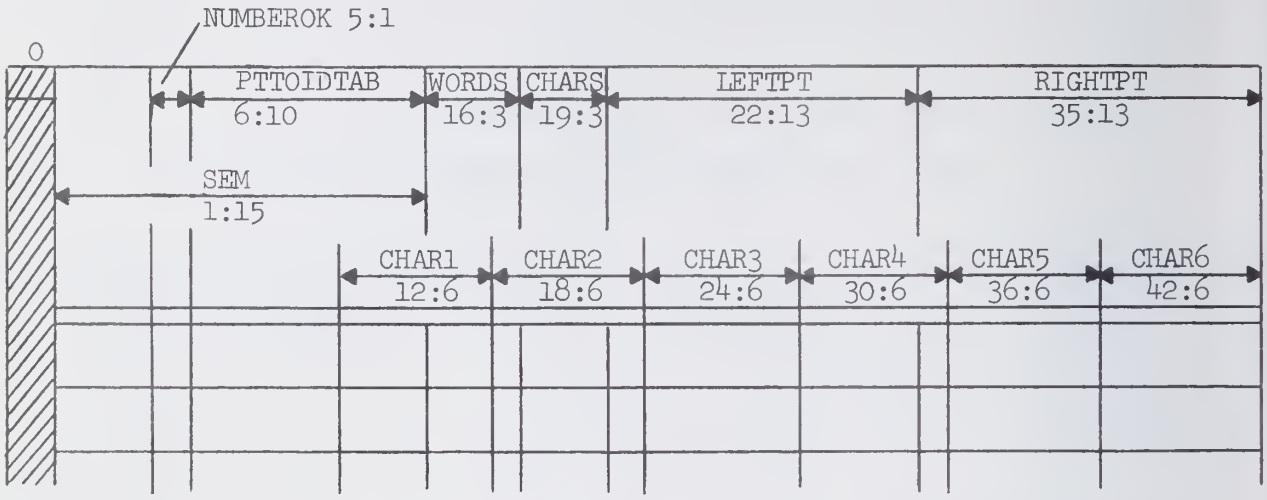
<forward action declaration> ::= $FORWRD(<list of action identifiers>)

<list of action identifiers> ::= <generalized identifier> |
<list of action identifiers> <generalized identifier>

Sometimes, it is possible to have in an ISL program actions which are never directly referred to in the syntax. Such actions are named "indirectly called" and, although never explicitly executed by the parser, may be called from other semantic actions, by means of the ISL execute statement.

The purpose of the forward action declaration is to inform the ISL translator of the future appearance of indirectly called actions. The translator must know this before the end of the outermost declaration part because, as explained in Section 4.1.3, ISLSWITCH must be generated containing one label for each action. The basic source of information for this is the file ACTIONS,

BIGTAB



MSTACK

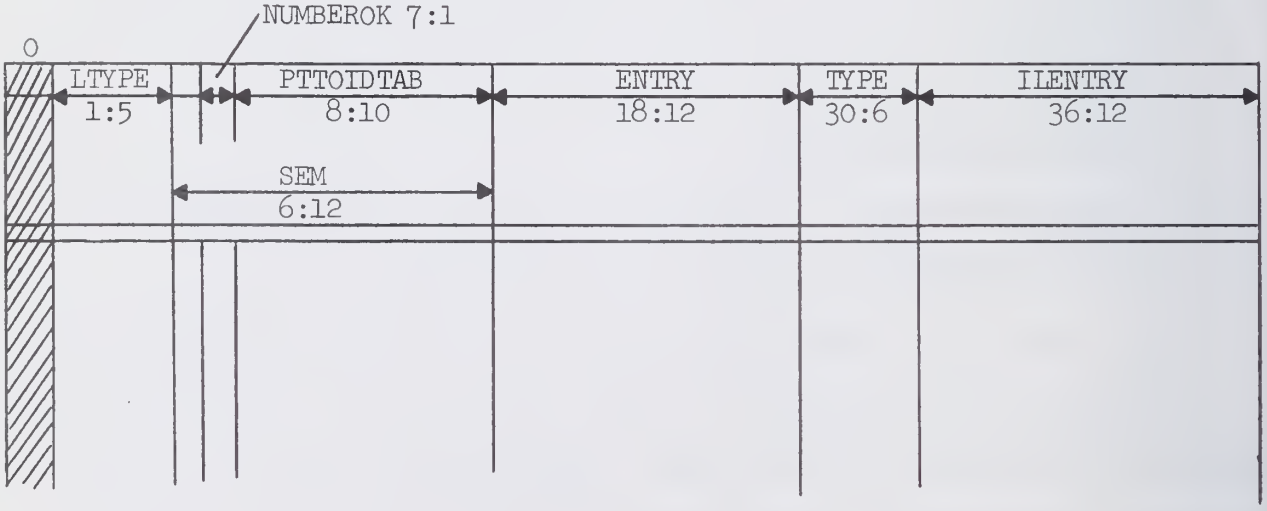


Figure 17. BIGTAB and MSTACK after an add column declaration.

but it does not contain the names of indirectly called actions since they never appeared in the syntax. Therefore, the forward action construct serves as a "declaration" of those actions. Obviously, forward action declarations must appear only in the outermost declaration part or else it is too late for the translator to use the information they provide.

An example of a forward action declaration is:

```
$FORWRD  (22, AUXIL)
```

This informs the translator that indirectly called semantic actions named "22" and "AUXIL" occur in the program.

4.2.2.2 Number of Actions Declaration

<number of actions declaration> ::= $UPPERN (<unsigned integer>)

The number of actions declaration is, like the forward action construct, related to providing information for the translator to build ISLSWITCH. In this case, however, this information is accepted in lieu of the file ACTIONS instead of in addition to it.

This declaration is needed only when there is no file ACTIONS available on disk at the time of the ISL translation. This might be the case when the user is simply debugging the semantics rather than trying to create a compiler. The unsigned integer in the declaration must then be equal to or larger than the total number of semantic actions present in the ISL program. The number of actions declaration enables the translator to create a dummy ISLSWITCH that allows the user to completely check the syntax of the resulting ALGOL code. Note, however, that the code created in this situation cannot be used in the final compiler since there was no file ACTIONS available to provide the matching of semantic action calls and declarations. When, at an ISL translation, the file ACTIONS is available, the number of actions declaration, if present, is

ignored. As with forward action declarations, number of action declarations must appear only in the outermost declaration part.

An example of a number of actions declaration is:

$UPPERN (22)

This informs the translator that there are no more than 22 semantic actions in the program and this information should be used to create a dummy ISLSWITCH if the file ACTIONS is not available.

4.2.2.3 In Declaration

<in declaration> ::= $ IN (<list of declarations>)

The purpose of this construct can best be understood by referring to Figure 14. There one can see that all declarations in the outermost declaration part of an ISL program are placed (in the translated ALGOL code) on the same level as the declaration of procedure EXEC, rather than internal to it. The reason for this is obvious: if the declarations were local to procedure EXEC then their contents would be lost each time EXEC is exited. Naturally, the user usually wants the contents of tables, stacks, etc. to be conserved between calls of semantic actions. Sometimes, however, it is necessary to insert declarations local to procedure EXEC: labels and switches to be used in the semantic actions, for example, must be declared local to EXEC due to BeA restrictions. This is the task of the in declaration. The <list of declarations> given as a parameter to this construct consists of a string of declarations separated by ";" This string is stored in a buffer and is later output local to procedure EXEC, following the declaration of ISLSWITCH. Obviously, there is no meaning in a nesting of in declarations so the <list of declarations> must not contain any $IN. Like the two previous constructs, the in declaration must appear only in the outermost declaration part.

An example of an in declaration:

$IN (LABEL L1 , L2 ; SWITCH SW ← L1 , L2)

The result of this construct is to place the declarations of L1, L2 and SW immediately following the declaration of ISLSWITCH and thus local to procedure EXEC.

4.2.2.4 End of Declarations Declaration

<end of declarations declaration> ::= $ENDEC

An end of declarations declaration is needed to mark the end of the declaration part of all inner blocks (i.e., all blocks internal to the outermost) if such declaration part contains at least one table or stack declaration. Although superfluous, an end of declarations declaration may also be present in any other block, including the outermost. Note, however, that when present the $ENDEC must appear as the last declaration just before the beginning of the compound tail. The purpose of $ENDEC is to enable the ISL translator to output initializations of table and stack pointers as the first statements of the translated compound tail. Without the $ENDEC a very sophisticated parsing would be needed to locate the beginning of the compound tail. A $ENDEC is not necessary in the outermost block because in such a block pointer initializations are inserted as the first statements of action 0 (see Section 4.1.2) and not at the beginning of the compound tail.

4.2.3 Define Declarations and Definition Calls

These constructs provide a simple macro-expander facility. The define declaration associates strings of characters with certain identifiers: the defined identifiers. The syntax is as follows:

<ISL define declaration> ::= $DEFINE (<ISL definition list>)

<ISL definition list> ::= <ISL definition part>

<ISL definition list>, <ISL definition part>

<ISL definition part> ::= <ISL defined identifier> = <definition> #

where a definition can be essentially any string of symbols not containing the character "#", which indicates the end of the definition.

An ISL define declaration does not cause any code to be output; it simply tells the ISL translator to store the string of symbols and associate it with the defined identifier. These identifiers then may be used in definition calls which have the following syntax:

<definition call> ::= $DF <ISL defined identifier>

In the production above the defined identifier must be concatenated to the $DF. A definition call is simply replaced at translation time by the string of symbols that has been associated with it by the define declaration.

ISL defines are basically an ISL implementation of BeA defines. ISL defines are expanded by the ISL translator while BeA defines are not recognized by the translator and will be obeyed only by the BeA compiler. Therefore, BeA defines may be used in all constructs that are transparent to the ISL translator while ISL defines must be used in constructs that will be parsed at ISL translation. Obviously the user might use only ISL defines for both types of constructs but since BeA defines are more convenient to use than ISL defines, he will probably use these only when the former would not work. The following example presents a case in which the ISL define could not be replaced by a BeA define.

Example of an ISL define declaration:

```
$DEFINE (LKLISTROWS = 256#,
        LKLISTCOLUMNS = PTTOABTAB = 1:8, LINK = 9:8#)
```

This definition might then be used in the following table declaration:

```
$TABLE (LKLIST, $DFLKLISTROWS, $DFLKLISTCOLUMNS)
```

which would be equivalent to the declaration of LKLIST given as an example in Section 4.2.1.1.Note that there is no restriction to the nesting of defines, i.e., a define declaration may contain definition calls.

4.3 ISL Operands

These constructs may be used in arithmetic expressions as primaries and enable the user to manipulate tables, stacks and cells. Operands are the second category of ISL constructs and they include three types of constructs:

a) the ISL operands;

b) pointer operands;

c) search operands.

4.3.1 The ISL Operand

This is a generic name for three operands: the table operand, the stack operand and the cell operand, all characterized by the string $OPD. This is by far the most common and useful ISL construct and simply designates one location of a table, stack or cell. It can be used either as a primary in an arithmetic expression or as the leftmost left hand side of an assignment statement. Used as a primary, a $OPD provides the value currently stored in the designated element of the table, stack or cell. As a left hand side of an assignment statement, it allows a value to be stored in the designated element.

The syntax for a table operand is as follows:

```
<table operand> ::= $OPD (<table identifier>, <row information>,
                    <column information>)
<row information> ::= <arithmetic expression> | $NOW | $NEXT | $TOP |
                      $TOP <adding operator> <arithmetic expression>
<column information> ::= <column identifier> | $ALL |
                         $ALL <arithmetic expression>
```

In order to designate an element of a table, a table operand must contain 3 pieces of information: the table name and the row and column of the element. Stack and cell operands are exactly similar to table operands except that <column information> does not appear in a stack operand and <row information> does not appear in a cell operand. Therefore only table operands will be explained in detail.

The row information of a table operand may contain either an arithmetic expression or a number of ISL constructs.An arithmetic expression appearing as row information designates simply that the row indicated is the row whose number is the value of the arithmetic expression. The special ISL constructs present an automatic way to make reference to the pointer associated with the table. In ISL, it is assumed that pointers point to the next available location of a table or stack.The top of MSTACK, for example, is pointed to by PTMSTACK-1. This is precisely the meaning of $TOP as row information; it is equivalent to: PT<table identifier> -1. $NOW is a symonym of $TOP kept for historical reasons. $TOP can be modified by an arithmetic expression and the normal use of this feature is to indicate the positions of a stack or table relative to the top. Thus $TOP - 1 points to the first position below the top and is equivalent to PT <table identifier> - 2. Finally $NEXT indicates the next available row and is therefore equivalent to PT <table identifier>.

The column information consists normally of a column identifier indicating in which column the designated element is to be found.

In special cases $ALL, or $ALL followed by an arithmetic expression, may be used as column information to indicate that the column is, in fact, the whole word. $ALL is used in tables with one word per row and is equivalent to a column defined as 1:47. $ALL followed by an arithmetic expression must be used in place of $ALL for tables with multi-word rows. The value of the arithmetic expression indicates which of the words of a row is to be picked. Thus $ALL 3 designates the column 97:47, i.e., all the third word of the row.

Examples of table operands:

$OPD(MSTACK, $NOW, ENTRY)

designates the location on top of MSTACK (i.e., pointed to by PTMSTACK-1), in the ENTRY FIELD. It is known that, for an identifier on top of MSTACK, the contents of this location is a pointer to the entry of the identifier in BIGTAB. Therefore, the first character of the BCD representation of the identifier could be picked up by the following operand:

$OPD (BIGTAB,$OPD(MSTACK,NOW,ENTRY) + 1, CHAR1)

Note the use of $OPD in the row information of another $OPD. This is very common in ISL and is very useful in following a structure of pointers down to several levels of indirect addressing.

4.3.2 Pointer Operand

<pointer operand> ::= $POINTER(<table identifier>) | $POINTER(<stack identifier>)

A pointer operand, which may be used as a primary or a left hand side of an assignment, yields the pointer associated with the table or stack; i.e., PT<table identifier> or PT<stack identifier>. Since it is easier and more efficient to use the name of the pointer directly, this construct is not normally used, although it can make an ISL program a little more readable for a user unfamiliar with ISL.

Example of a pointer operand:

$POINTER(MSTACK)

This is equivalent to: PTMSTACK

4.3.3 Search Operand

This operand can only be used as a primary in arithmetic expressions. There are 2 types of search operands: stack searchs and table searchs. Since the stack search operand is similar to the table search operand, except for the absence of <column information>, only the table search will be treated here. The syntax for a table search operand is the following:

<table search operand> ::= $SEARCH(<table identifier>,
<column information>,
<arithmetic expression>)

This operand performs a sequential search on the table specified, starting at the current top of the table (i.e., PT<table identifier>-1) and going down (i.e., to lower numbered rows) by successive rows. For each row, the contents of the field designated by the column information is compared with the value of the arithmetic expression which is the 3-rd parameter of the operand. The operand returns the row number of the first row for which the comparison above is successful. If there is no match, the value returned is -1.

Example of a table search operand:

$SEARCH(MSTACK,TYPE,3)

The value returned will be the row number of the "first" (in the sense above) row of MSTACK that contains a string (i.e., type 3). If no row of MSTACK contains a string, the value returned is -1.

4.4 ISL Statements

ISL Statements are used just like BeA statements and provide a "higher level" degree of manipulation of tables, stacks and cells as compared with the manipulations allowed by ISL operands. The main statements enable the user

to push and pop stacks or tables, enter information in a table or cell and to emit intermediate code. There are also the increment and decrement statements and the execute statement. Each statement is now individually analyzed.

4.4.1 Increment and Decrement Statements

<increment statement> ::= $INCR(<leftmost left part>)

<decrement statement> ::= $DECR(<leftmost left part>)

where a leftmost left part is any construct allowed as the leftmost left hand side of an assignment statement. These statements simply generate code to add one to the leftmost left part ($INCR) or to subtract one from it ($DECR). Although they perform a trivial task, they can contribute to the readibility of an ISL program.

Example:

```
$INCR(PTABTAB)
```

which is equivalent to:

```
PTABTAB ← PTABTAB + 1
```

4.4.2 Execute Statement

<execute statement> ::= $EXEC(<generalized identifier>)

The purpose of this statement is to allow the user to request, in a given semantic action, the execution of another action whose name is the only parameter of the construct. Thus, if an action called "22" contains a subset of what must be done in action "7", there is no need to repeat "22" in "7"; at the appropriate point in "7", "22" may be called for execution with the following statement:

```
$EXEC(22)
```

The same effect could be obtained with procedures (and, in fact, $EXEC simply causes a recursive call of the procedure EXEC) but the use of $EXEC seems to be simpler and more natural.

4.4.3 Push and Pop Statements

These dual statements allow manipulation of information in a row of a table or stack with automatic adjustment of the pointer. For both statements there is a table form and a stack form. Since stack pushes and pops are a trivial simplification of table push and pop statements, only the latter will be considered here. The syntax for stack push and pop statements is presented in Appendix A, productions 62 and 67.

<table push statement> ::= $PUSH(<table identifier>,<entry list>)

<entry list> ::= <entry element> | <entry list>, <entry element>

<entry element> ::= <column information> : <arithmetic expression>

<table pop statement> ::= $POP(<table identifier>, <pop entry list>) |

$POP(<table identifier>)

where a pop entry list is exactly similar to an entry list except that instead of arithmetic expressions the pop entry list must contain leftmost left parts.

The effect of a $PUSH is the following: the next available row in the table (which is pointed to by the pointer associated with the table) is taken and the field corresponding to each column information in the entry list is iniatilized to the value of the arithmetic expression that follows the column information. Any field not referred to by the entry list is left unchanged. Then the pointer associated with the table is incremented to point to the next row.

A $POP is essentially the dual of a push statement. Its effect is the following: the present top row of the table (which is pointed to by PT<table identifier>-1) is taken and each leftmost left part in the pop entry

list is initialized to the value stored in the field referred to by the column information that precedes the left most left part. Then the pointer associated with the table is decremented to point to the row just used. Notice that no information is erased from the table, but the row is now marked as available and eventually will be overwritten.

There is one simple form of $POP which does not contain a pop entry list. The effect of this is simply to decrement the pointer.

Examples of push and pop statements:

```
(a)  $PUSH(IDTAB,PTTOBIGTAB : TEMP2, BACKPT : TEMP1,
                 PTTOABTAB  : PTABTAB, LEVEL : CULEVEL,
                 TYPE : STORETYPE, REST: 0)
```

which is equivalent to the following compound statement:

```
BEGIN
     $OPD(IDTAB,PTIDTAB,PTTOBIGTAB) ← TEMP2;
     $OPD(IDTAB,PTIDTAB,BACKPT    ) ← TEMP1;
     $OPD(IDTAB,PTIDTAB,PTTOABTAB ) ← PTABTAB;
     $OPD(IDTAB,PTIDTAB,LEVEL     ) ← CULEVEL;
     $OPD(IDTAB,PTIDTAB,TYPE      ) ← STORETYPE;
     $OPD(IDTAB,PTIDTAB,REST      ) ← 0;
     $INCR(PTIDTAB)
END
```

```
(b)  $POP(IDTAB)
```

which is equivalent to $DECR(PTIDTAB)

4.4.4 Enter Statement

The enter statement is a generalization of the table push statement and is used to initialize several columns of a cell or a row of a table with

only one statement. Consequently, there is a table enter statement and a cell enter statement. Because of similarities between these, only table enter statements will be discussed here.

<table enter statement> ::= $ENTER(<table identifier>,
<row information>,
<entry list>)

The effect of a $ENTER statement is precisely the same as the effect of a $PUSH except for the two following differences:

a) instead of always picking the next available row of the table, the enter statement utilizes the row designated by the row information;

b) no pointer increment is performed by an enter statement.

Example of an enter statement:

$ENTER(IDTAB,TEMP1,PTTOLKLIST : PTLKLIST-1, USED : 1)

which is equivalent to the following compound statement:

```
BEGIN
    $OPD(IDTAB,TEMP1,PTTOLKLIST) ← PTLKLIST-1;
    $OPD(IDTAB,TEMP1,USED) ← 1
END
```

4.4.5 Code Statement

<code statement> ::= $CODE(<list of operators and operands>)

<list of operators and operands> ::= <operand> | <operator> |
<list of operators and operands>,<operand> |
<list of operators and operands>,<operator>

<operand> ::= <arithmetic expression>

<operator> ::= <static operator> | <dynamic operator>

<static operator> ::= * <generalized identifier>

<dynamic operator> ::= # <arithmetic expression>

The $CODE statement provides output of IL operators and operands. Each element in the list of operators and operands causes the writing of one element in the intermediate language string.

If the element is an arithmetic expression, an IL operand will be emitted. The rightmost 16 bits of the value of the expression are taken as the IL element. A check is performed, however, to assure that an IL operand will result: field 32:4 of the value of the expression, which will be the IL table number, is tested and, if 0, is replaced by a 1. Thus if the user tried to output an operand with a table number = 0 (which would make it an operator), a table number = 1 is arbitrarily inserted.

If the element of the list of operators and operands is an arithmetic expression preceded by the character "#", then an IL operator will be emitted. The rightmost 12 bits of the value of the expression are taken as the IL entry and the IL element is completed with the addition of a table number field = 0, thus making sure that an IL operator will result. Note that the arithmetic expression that defines the operator is evaluated each time the $CODE statement is executed in a semantic routine. Therefore, the operator emitted may depend on a dynamic condition such as the situation of MSTACK. In fact, this is one of the most common uses of a dynamic operator as shown by the following example:

```
$CODE(#$OPD(MSTACK,PTMSTACK-2,ENTRY))
```

This statement emits, when executed, an IL operator whose code number (i.e., the IL entry number) is equal to the current value of the entry field in TOP-1 of MSTACK. Therefore, the same statement could be used to emit code when any of the following productions is reduced:

<BOOLEAN EXPRESSION> ::= <BOOLEAN EXPRESSION> #OR <BOOLEAN TERMS>

<BOOLEAN TERM> ::= <BOOLEAN TERM> #AND <BOOLEAN PRIMARY>

<TERM> ::= <TERM> × <ARITHMETIC PRIMARY>

<TERM> ::= <TERM> / <ARITHMETIC PRIMARY>

The operator emitted would be different in each case and equal to the code number associated with the terminals "AND", "OR", "×" and "/", respectively (see Section 2.2.1.1 for an explanation of the assignment of code numbers to terminals). It is known that each semantic action has a code number associated with it. Thus, if the code numbers of the semantic actions in pass 2 were equal to the code numbers associated by the scanner, an automatic "linking" would result, i.e., only one code statement in pass 1 semantics allows the system to execute in pass 2 one of four pass 2 semantic actions named: "AND", "OR", "×", and "/". This also is the reason for allowing, as semantic action names strings containing one single character. The way provided for automatic linking of terminal code numbers with pass i (i>1) semantic action code numbers will be described as static operators are explained.

If an element of the list of operators and operands is a generalized identifier preceded by the character "*", then an IL operator will be emitted. In this case, however, the operator emitted will always be the same for every execution of the code statement. This is why this type of operator emission is called static.

EXAMPLE: #CODE(*GETSPACE)

This statement emits an IL operator containing as entry number a code number associated (by the ISL translator) with the operator named "GETSPACE". The method used to associate this code number is the following: The ISL translator keeps a table called SPSTAB which contains a list of generalized identifiers and associated code numbers. This table is initialized,

at the beginning of the ISL translation, according to the setting of the option SPSTAB which is described in Chapter 5. Thus, when a static operator is found, the operator name is looked up in SPSTAB. If found, the associated code number is picked and emitted as the entry field of the operator. If not found, the operator name is entered in SPSTAB and the next available code number is associated with it; then the IL operator is emitted. Therefore, at the end of the pass i semantics translation, an updated SPSTAB is available and is used to generate two files which may be utilized by the semantics of the next pass: the files ACTIONSi and OPRTABi. Thus, for pass i (i>1), the file ACTIONS needed by the ISL translator is provided by translation of the previous pass while for pass 1 it is provided by the TWINKLE translator. The complete file structure is detailed in Chapter 5.

The method provided for linking terminal code numbers with semantic action code numbers is, thus, automatically provided by one setting of the SPSTAB option which causes SPSTAB to be initialized with the list of terminal names and their code numbers.

5. USE OF THE ISL TRANSLATOR

This chapter is dedicated to describing details of the actual use of the ISL translator. Besides a description of the file structure and instructions for running the translator, it contains a detailed discussion of its control cards and control options. A good understanding of ISL control options is essential to obtaining the full advantage of the automation that they provide.

5.1 The File Structure

For a description of the set of files used by the ISL translator, it must be initially remarked that files on the B-5500 disk are characterized by a pair of identifiers (each with no more than seven characters) separated by a slash:

<file name> ::= <prefix> / <suffix>

In the implementation of a given language with the system, a name must be picked for the language. The language name (henceforth called ℓ.n.) is input to both the TWINKLE and ISL translators and all the files created by the system for that language will have a name of the form: <ℓ.n.> / <suffix> . Thus several different languages implemented on the system may coexist on the B-5500 disk.

5.1.1 Block Diagram

Figure 18 presents a diagram of all the files that may be involved in an ISL translation. It is assumed that the semantic program being translated is the description of pass i. The following is a discussion of each file indicated in Figure 18.

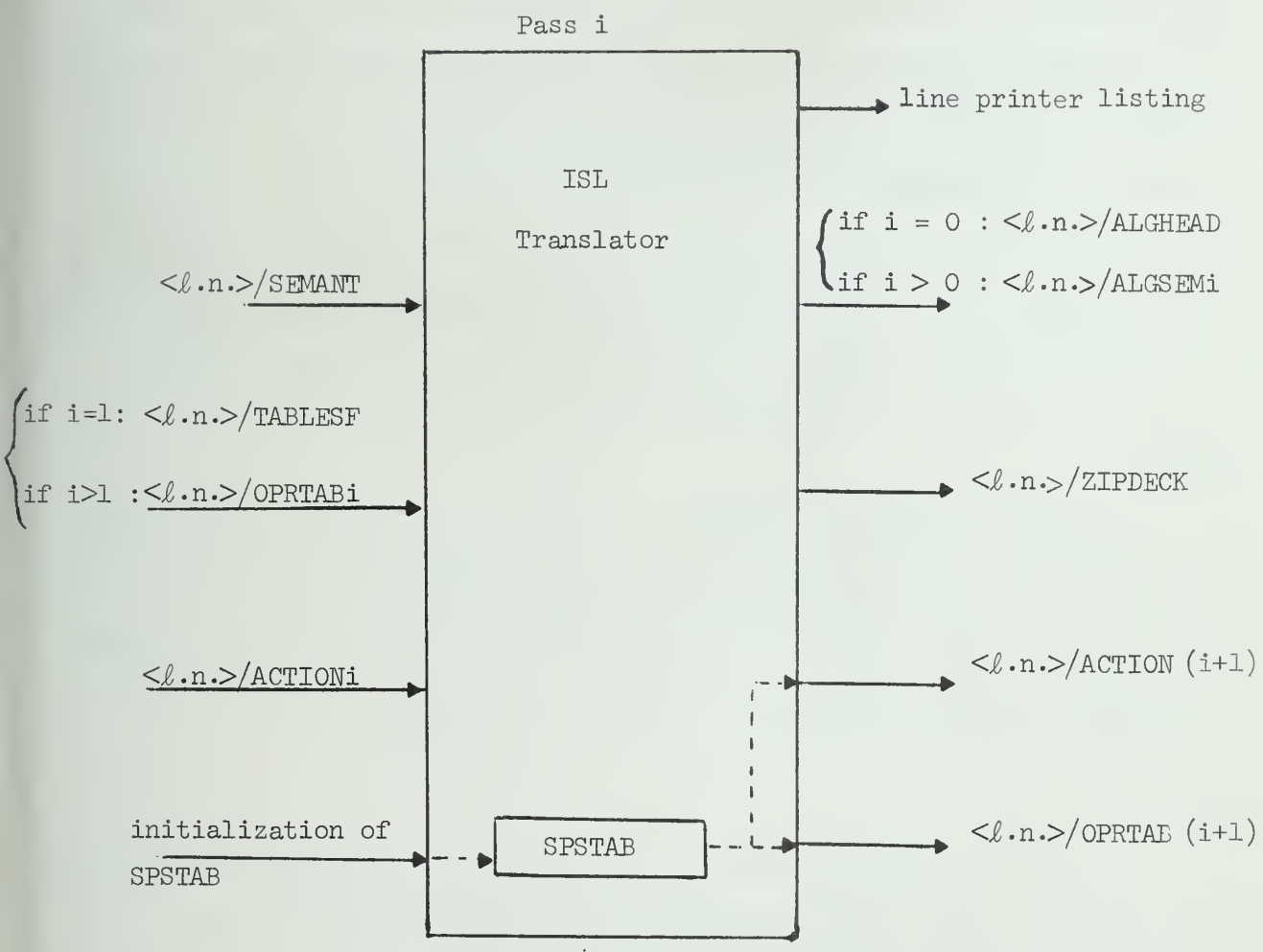


Figure 18. Diagram of the files used by the ISL translator.

SEMANT:

This is the primary input file for the ISL translator. It contains the source code in ISL card images as well as control cards. The structure of the file SEMANT is illustrated in Figure 19. There one can see that SEMANT is in fact composed of a number of semantic programs, each identified by the option PASSNUMBER (see section 5.3.4). Obviously, for the translation of pass i only the semantic program identified as PASSNUMBER i is taken as input. For i = 0, SEMANT is the only input file used and its PASSNUMBER 0 must be an ISL program containing only a list of declarations (i.e., the compound tail of the outermost block must be: $END). Also the only file generated is <ℓ.n.> / ALGHEAD.

OPRTABi:

This file basically contains, in a binary tree structure, the action names present in the file ACTIONi. It is used to allow efficient lookup of the action names. For i = 1 this tree is in the table OPRTAB of the file TABLESF. Thus TABLESF is used in this case.

ACTIONi:

This file contains a list of action names and the associated code numbers. Note that for i = 1 the suffix ACTIONS is used in lieu of ACTION1. Also, for i = 1, the names correspond to semantic action calls found in the syntax and the file is created by the TWINKLE translator. For i > 1, the names correspond to IL operators found in $CODE statements in the previous pass and the file is created by the ISL translation of such previous pass.

initialization of SPSTAB:

SPSTAB (see section 4.4.5) can be initialized in a number of different ways, depending on the setting of the option SPSTAB as described in section 5.3.6.

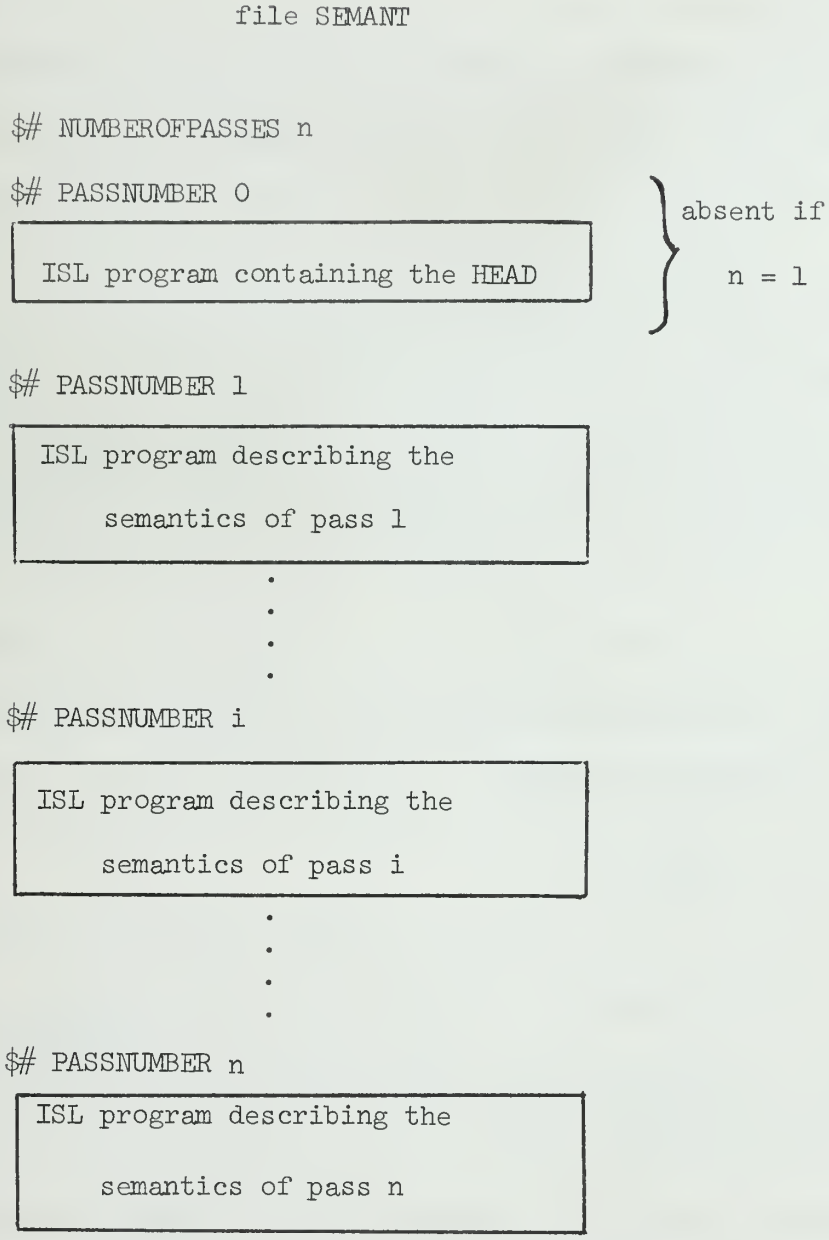


Figure 19. Structure of the file SEMANT.

line printer listing:

If the proper option is set (see section 5.3.2) the ISL translator outputs a listing containing, side by side, the source cards and the corresponding output cards. Appendix E presents a sample of such a listing.

ALGSEMi:

This is a file of BeA card images containing the result of the translation. The structure of this file is detailed in Figure 14. Note that, for i = 0, the suffix ALGHEAD is used in lieu of ALGSEMO. Also, the structure of ALGHEAD is much simpler than the structure of ALGSEMi; ALGHEAD is a straight translation of HEAD, without the addition of any declarations.

ZIPDECK:

This file is created when the option SAVEZIPDECK is set. See section 5.3.5 for a description of ZIPDECK.

ACTION (i + 1) and OPRTAB (i + 1):

At the end of the translation, SPSTAB contains the names of the operators found in $CODE statements and their respective code numbers. This information is used to generate the files OPRTAB and ACTION for the next pass, i.e., OPRTAB (i + 1) and ACTION (i + 1).

It should also be noticed that the file OPRTABi, besides being used at the translation of pass i, is also needed at compile time if, at pass i, the IL is to be listed. This allows the printing of the name of each IL operand, thus making debugging much easier.

5.1.2 Selection of the Input File

It has already been mentioned that the file ACTIONS (i.e., ACTION for pass 1) is built by the TWINKLE translator and contains basically a list of the semantic action names that appeared in the syntax along with their code numbers.

In addition to this, two other pieces of information may also be present in ACTIONS:

a) TWINKLE allows the ISL code for an action to be placed in the syntax itself, following the action call. These actions are called intrinsic actions and the code that defines them is put in the file ACTIONS following their names and code numbers.

b) As described in section 2.3.2, the user may decide to input the semantic description of the language (i.e., the file SEMANT) to the TWINKLE translator, following the syntactic description. The semantics is then placed at the end of the file ACTIONS, as a tail to it.

Therefore, the ISL translator must decide whether to read directly its main input file (SEMANT) or whether to take it from the tail of ACTIONS. Also the code describing intrinsic actions must be removed from ACTIONS and inserted in SEMANT at the appropriate place. Figure 20 shows how these decisions are taken.

5.2 Running the Translator

This section contains specific instructions about how to run the ISL translator on the B-5500. Initially, one should keep in mind that the translator can be run in two different ways: directly or indirectly. Each of these is now described separately.

5.2.1 Indirect Run

This type of run is obtained when the user specifies, using TWINKLE control options, that he wants the TWINKLE translator to automatically initiate the ISL translator. The way to specify this is described by Mercer [3]. In this mode the TWINKLE translator communicates the ℓ.n. to the ISL translator.

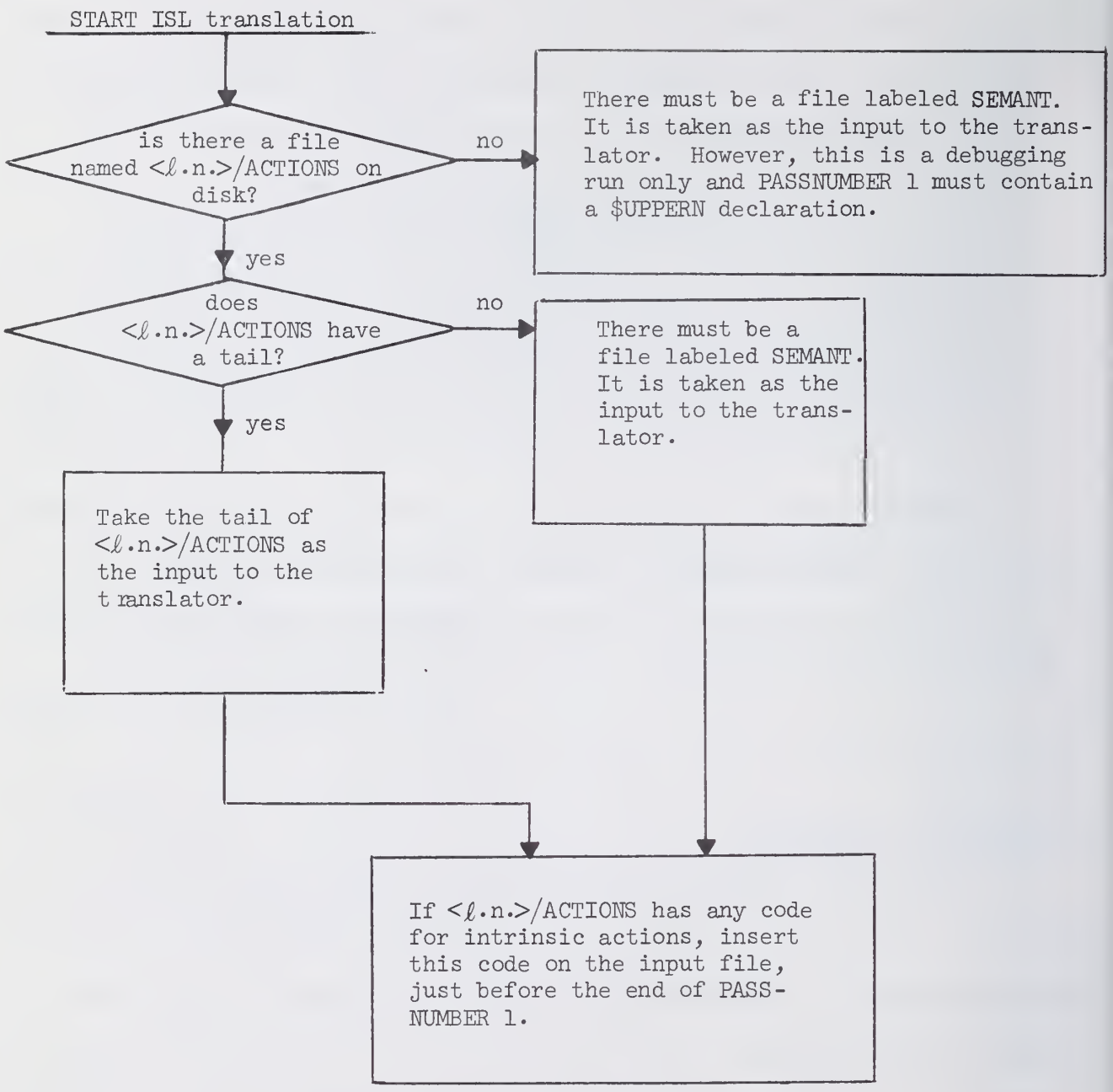


Figure 20. Flow diagram of the selection of the input file.

If all the files expected by the ISL translator are available, the user does not have to interfere.

5.2.2 Direct Run

For debugging the semantic programs it may be more convenient to run the ISL translator directly. This can be done by submitting, to the B-5500, one of the control decks illustrated in Figure 21. Note that in this case the user must communicate *ℓ*.n. to the ISL translator. This is done by means of the first identifier in the COMPILE card. The second identifier is completely irrelevant and is only used in the heading of the listing produced by the translator. Note also that either the word LIBRARY or the word SYNTAX must be present at the end of the COMPILE card. The normal mode of operation is with the word LIBRARY. SYNTAX is used for syntax checks only and no result whatsoever is kept on disk in this type of run.

5.3 ISL Control Cards and Control Options

Control cards may appear at any place in the input string for the ISL translator. They are completely free-field except for a "$" in column 1 and a "#" in column 2. Blanks (one or more) should be used as separators between the options. There is no provision for continuation control cards. In the improbable case in which all the desired options do not fit on one card (columns up to 71 can be used) a new control card can be submitted containing the remaining options. There is no limit on the number of ISL control cards used. Each control card affects only the options explicitly mentioned in it, all the other options remaining unchanged.

ISL control cards can be syntactically described as:

<ISL control card> ::= <ISL control card prefix> |
<ISL control card> <option>

a) File SEMANT is on cards

```
?USER = <user code>
?COMPILE  <ℓ.n.>/<identifier> WITH ISL { SYNTAX
                                       { LIBRARY
?DATA CARD

┌──────────────────────────┐
│ deck of cards containing │
│     the file SEMANT      │
└──────────────────────────┘

?END
```

b) File SEMANT is on disk under the name <prefix>/<suffix>

```
?USER = <user code>
?COMPILE  <ℓ.n.>/<identifier> WITH ISL { SYNTAX
                                       { LIBRARY
?ISL FILE CARD = <prefix>/<suffix> SERIAL
?END
```

c) File SEMANT is the tail of <ℓ.n.>/ACTIONS

```
?USER = <user code>
?COMPILE  <ℓ.n.>/<identifier> WITH ISL { SYNTAX
                                       { LIBRARY
?END
```

Figure 21. Control decks for direct runs of the ISL translator.

<ISL control card prefix> ::= {dollar sign ("$") and sharp sign ("#") in columns 1 and 2 of the card, respectively}

Options: Only the first five characters of an option identifier are analyzed by the ISL translator so any option identifier containing more than five characters can be truncated to its first five characters.

<option> ::= LANGUAGE <string> | VERSION <unsigned integer> |
LIST | NOLIST | ALGSEMLISTONLY | NOALGSEMLISTONLY |
COMPLIST | ALLIST | SEQ <sequence parameters> | NOSEQ |
NEWSEQ | NONEWSEQ | SUBINC <unsigned integer> |
NUMBEROFPASSES <unsigned integer> |
PASSNUMBER <unsigned integer> <where it is> | USEIL |
NOUSEIL | TEST | SPECIAL <list of special parameters> |
SAVEZIPDECK <where to save> | INTERPRETIVE |
SPSTAB <spstab parameter>

<sequence parameters> ::= <empty> | <unsigned integer> |
+ <unsigned integer> |
<unsigned integer> + <unsigned integer>

<where it is> ::= <empty> | ONDISK | ONDISK : <prefix> / <suffix>

<list of special parameters> ::= <special parameter> |
<list of special parameters> ,
<special parameter>

<special parameter> ::= SCANNER | PARD1 | FILLTAB | PARD2 | PBEGIN |
PEND | LK | FPI | FPF | ILRECOG

<where to save> ::= DISK | PUNCH

<spstab parameter> ::= EMPTY | CHARACTERS | TERMINALS | OPRTAB | ONDISK

5.3.1 The Language Options

Two options fall in this group: LANGUAGE and VERSION. Compilers created by the system print a heading line on the listings containing the following information:

XXXXXXXXXXX COMPILER - VERSION n - <date and hour>

The option LANGUAGE provides a way for the user to specify what he wants printed in place of the string of X's above. The contents of the string that follows the option are picked out as the characters to be printed in the heading. The string may contain up to twelve characters; if it contains less, they are right justified in a field of blanks.

Example: LANGUAGE "DEMALGOL - I"

Default: The language name known by the ISL translator (see section 5.2).

The option VERSION provides a way to insert the integer n that is printed in the heading.

Example: VERSION 2

Default: VERSION 1

5.3.2 The Listing Options

This group includes the following options: LIST, NOLIST, ALGSEMLISTONLY, NOALGSEMLISTONLY, COMPLIST and ALLIST. The discussion of the last two will be postponed until section 5.3.5.

Options LIST and NOLIST:

These options set ON or OFF, respectively, the Boolean variable LLIST which controls listing on the line printer. If LLIST is OFF, no listing will be performed on the line printer. However, if an ISL error is found and LLIST is OFF, an error message, the last card read, and the contents of the output buffer will be printed. If LLIST if ON, system messages and the output cards (i.e., ALGSEM) will be listed. The input cards will also be listed if ALGSEMLISTONLY is not set.

Default: LIST

Options ALGSEMLISTONLY and NOALGSEMLISTONLY:

These options set ON or OFF, respectively, the Boolean variable LISTOUTONLY. If LLIST is OFF, the state of LISTOUTONLY is irrelevant. If LLIST is ON and LISTOUTONLY is OFF, a complete listing will be output containing system messages, input and output cards. If LLIST is ON and LISTOUTONLY is ON, the input cards will not be listed and the rest of the listing remains unaffected by the state of LISTOUTONLY.

Default: NOALGSEMLISTONLY

5.3.3 The Sequencing Options

These options are related to assigning a sequence number to the cards of ALGSEM. They include: SEQ, NOSEQ, NEWSEQ, NONEWSEQ and SUBINC.

Options SEQ and NOSEQ:

These options set ON or OFF, respectively, the Boolean variable SEQ which controls the assignment of sequence numbers to the card images in ALGSEM. If SEQ is OFF, the sequence field (columns 73 - 80) of the card images will be left blank. If SEQ is ON, the sequence field will be filled with a sequence number. The value of this sequence number is determined by the <sequence parameters> which may follow the identifier SEQ, by the value of

SUBINC, by the sequence number in the last input card read and by the state (ON or OFF) of NEWSEQ.

If NEWSEQ is OFF, the sequence number of an output card image will be the sequence number of the last input card read. However, when several output cards are generated after the reading of only one input card, the first output card will have the same sequence number as the input card, say m. The second output card will have the sequence number m + SUBINC . . . the n-th output card will have the sequence number m + (n - 1)SUBINC. No test is performed to prevent assigning the same sequence number to more than one output card; however, since it is extremely improbable that more than a few dozens of output cards are generated without reading a new card, one can be reasonably sure than no duplicate sequence number will be obtained if SUBINC is small (say 1 or 2) and the input cards are sequenced with a large increment (say 100).

If NEWSEQ is ON, the sequence field of the input cards is irrelevant and sequence numbers will be assigned to the output cards as follows: The first sequence number will be the value of the integer variable INITIALCARDNUMB and the increment will be the value of the integer variable INC. Both INC and INITIALCARDNUMB are initially set to 100. These values can be modified, if desired, by using a non-empty sequence parameter part after the identifier: SEQ. A sequence parameter part containing only an unsigned integer will set INITIALCARDNUMB to the value of that integer. A sequence parameter part containing + <unsigned integer> will set INC to the value of the integer. Finally, a sequence parameter type of the form: <unsigned integer> + <unsigned integer> will set both INITIALCARDNUMB to the value of the first integer and INC to the value of the second integer.

Default: SEQ 100 + 100

Options NEWSEQ and NONEWSEQ:

These options set ON or OFF, respectively, the Boolean variable NEWSEQ whose role has been described above.

Default: NONEWSEQ

Note that when NEWSEQ is OFF (as in the default setting), the card images in SEMANT are expected to have a sequence number, as described. If the sequence field of a card in SEMANT is blank, the sequence number 00000000 will be assumed.

Option SUBINC:

This option, which must be followed by an unsigned integer, sets the value of the variable SUBINC to the value of the unsigned integer. The use of SUBINC was described previously.

Default: SUBINC 1

5.3.4 The Multi-pass Options

These options are related to regulating the generation of multi-pass compilers. They are: NUMBEROFPASSES, PASSNUMBER, USEIL and NOUSEIL.

Option NUMBEROFPASSES:

This option defines the integer that must follow it as the number of passes used by the compiler. It is mainly used for checking purposes and to decide whether or not there must be a file <ℓ.n.> / HEAD. The translator expects to find this file on disk when NUMBEROFPASSES is greater than one and the first semantic program input to the compiler had a non-zero PASSNUMBER.

Example: NUMBEROFPASSES 3

Default: NUMBEROFPASSES 1

Option PASSNUMBER:

PASSNUMBER assigns the integer that must follow it as the number of the next semantic program to be input to the translator. If NUMBEROFPASSES > 1, PASSNUMBER 0 means the HEAD. In an indirect run, all passes must be present in sequence. Therefore, the first semantic program is assumed to be #0, and so on in sequence. If PASSNUMBER options are present, they are used just as a check. In direct runs, the option PASSNUMBER is followed blindly and it is the responsibility of the user to "label" each semantic program with the correct PASSNUMBER option. If the word ONDISK follows the integer in a PASSNUMBER option it indicates that the corresponding program is on disk (under the name <prefix> / <suffix>) instead of being the next set of cards. The translator will then read the program from disk. If no <prefix> / <suffix> follows the word ONDISK, the following names are expected in the disk files: for PASSNUMBER 0, <ℓ.n.> / HEAD; for PASSNUMBER i (i > 0), <ℓ.n.> / SEMANTi.

Example: PASSNUMBER 2

Default: As explained above.

Options USEIL and NOUSEIL:

The option USEIL must be used whenever an IL string is created in a given semantic program. This option guarantees that the IL file will be kept on disk for the next pass of the compilation. No file of IL will be saved on disk if the option NOUSEIL is in effect.

Example: USEIL

Default: NOUSEII

5.3.5 The Zipping Options

These options are related to the automatic initiation of the BeA compiler to compile the compiler. The verb ZIP is used, in the B-5500 system, to denote the initiation of one program by another and will be used here with this

meaning. It has been mentioned in Chapter 2 that the ISL translator mounts the ALGOL source for the compiler by putting together several pieces of ALGOL code. These pieces are detailed in Figure 10. The ISL translator performs this mounting by constructing a file with "calls" to the appropriate pieces of code and submitting this file (called ZIP file) to the BeA compiler by means of a ZIP statement. The ALGOL compiler understands these calls and picks the pieces out itself. Two different types of calls are used in the ZIP file. The first type is of the form:

$$ C <piece name>

where the piece name is an identifier. This type of call is used to extract the pieces of code from the TWS Symbolic Library. For instance, the call

$$ C SCANNER

inserts the SCANNER taken out of the TWS Symbolic Library which is kept on the B-5500 disk under the name TWS/FILES. The second type of call is of the form:

$$ D <prefix> / <suffix>

and causes the insertion of the disk file named <prefix> / <suffix> . This type of call is used to place the language-dependent parts like the files <ℓ.n.> / ALGSEMi . It is now possible to understand Figure 22, which presents the ZIP deck in detail. Notice that the form of the ZIP deck depends on the combination of executable and interpretive pieces required by the user. This information is fed via TWINKLE control options and is kept on TABLESF from where it is picked out by the ISL translator when it is time to build the ZIP deck.

Option TEST:

This option, when set, inhibits the zipping to the BeA compiler at the end of an ISL translation. Once this option has been set, there is no provision

```
?COMPILE  <ℓ.n.>/DISK ALGOL LIBRARY
?ALGØL FILE CASTC = TWS/FILES SERIAL;DATA CARD
BEGIN

  $$ C GLOBAL

  $$ D <ℓ.n.>/ALGHEAD

  BEGIN

    $$ C SCANNER

    $$ C PAR1DEC

    if int. FP:  DEFINE IFP = TRUE #;
    if ex.  FP:  DEFINE IFP = FALSE #;

    if int. LK:  DEFINE ILK = TRUE #;
    if ex.  LK:  DEFINE ILK = FALSE #;

    if int. FT:  $$ C FILLTAB
    if ex.  FT:  $$ D <ℓ.n.>/FILLTAB

    $$ C PAR2DEC

    $$ C PBEGIN

    $$ C PEND

    if int. LK:  $$ C LK
    if ex.  LK:  $$ D <ℓ.n.>/LK

    if int. FP:  $$ C FPI
    if ex.  FP:  $$ C FPE

    if int. FP:  blank card
    if ex.  FP:  $$ D <ℓ.n.>/PARSER

    $$ D <ℓ.n.>/ALGSEM1

  END

  BEGIN

    $$ C ILRECOG

    $$ D <ℓ.n.>/ALGSEM2
       :
    $$ D <ℓ.n.>/ALGSEMn

  END

END.

?END
```

int. FP means interpretive FP package

ex. FP means executable FP package

int. LK means interpretive procedure LK

ex. LK means executable procedure LK

int. FT means interpretive procedure FILLTAB

ex. FT means executable procedure FILLTAB

Figure 22. The ZIP deck for the compilation of the compiler.

to reset it. Note that the zipping is also inhibited if any errors have been detected by the ISL translator.

Default: TEST not set (i.e., the system will try to zip to the BeA compiler).

When the zipping is performed, the compilation will run under the name ALGOL/<ℓ.n.> and, if successful, will enter on disk the file <ℓ.n.>/DISK which is the final product of the system. Instructions about its use are given in [4]. Note that, although it is always the ISL translator that creates the ZIP file, it does not always perform the ZIP. In fact, when executable pieces are used, the ALGOL Code Generator must create those pieces and the zip is then performed (with the ZIP deck created by the ISL translator) either by the ALGOL Code Generator or by the ISL translator, whichever finishes last.

Option INTERPRETIVE:

This option, when set, causes the ISL translator to ignore the information in TABLESF about which pieces are executable and to create a ZIP deck for all interpretive pieces. This is useful when one is simply trying to debug the semantics and the executable pieces are not available. Once this option has been set, there is no provision to reset it.

Default: INTERPRETIVE not set.

Option SAVEZIPDECK:

When a zip is performed with the ZIP deck, the deck is automatically removed from disk. If the user wants to save a copy of the ZIP deck for future direct use, he can set the option SAVEZIPDECK and the deck will be saved: on disk under the name <ℓ.n.>/ZIPDECK if <where to save> is DISK, and on punched cards if <where to save> is PUNCH.

Default: SAVEZIPDECK not set.

Option SPECIAL:

The system normally uses, for the non-language-dependent pieces, its own standard files which are kept in the TWS Symbolic Library. However, sometimes the user needs special features in one or more of these pieces and cannot use the standard ones. In this case the option SPECIAL should be used; it means that the system will pick each piece listed in the <list of special parameters> out of disk, under the name <ℓ.n.>/<piece name>. Thus the option SPECIAL replaces $$ C <piece name> calls by $$ D <ℓ.n.>/<piece name> . It is the user's responsibility to provide the special pieces (which usually are the standard ones with minor additions and modifications) and insert them on disk. Once SPECIAL has been set, there is no provision to reset it.

Default: SPECIAL not set.

Options COMPLIST and ALLIST:

These options are related to listing during the ALGOL compilation of the compiler. In this compilation, listing is always turned off for all blocks except the ALGSEM blocks, since these are the only parts the normal user might be interested in listing. Listing of the ALGSEM blocks only can be turned on by means of the option COMPLIST. In the event complete listing of the compilation is desired, the option ALLIST can be used. Once either of these options is turned on, there is no provision to turn it off. Note, however, that listing of parts of ALGSEMi can be obtained using ALGOL $ cards in the semantic programs since these cards are recognized and passed to ALGSEMi at ISL translation time. ALGOL $ cards may also be introduced anywhere in the ZIP deck.

Both COMPLIST and ALLIST turn on not only the listing but also the printing of the PRT locations.

Default: COMPLIST off and ALLIST off (i.e., no compilation listing at all will be provided).

5.3.6 The Option SPSTAB

This option provides a choice between five different types of initialization performed on SPSTAB, whose role was explained in section 4.4.5. Each SPSTAB parameter is now discussed:

EMPTY: No initialization is performed in SPSTAB, therefore the first IL operator will receive code number 0, the second code number 1 and so on. This parameter is used when no dynamic IL operators have been emitted with $CODE statements thus no linkage is needed with the code numbers of the IL operators.

CHARACTERS: SPSTAB is initialized with the character set and their internal code numbers. This is useful when dynamic IL operators expect to find only characters as actual parameters.

TERMINALS: SPSTAB is initialized with the complete set of terminals and their code numbers. This parameter is probably the most useful since it provides the linkage between terminal code numbers and semantic action code numbers mentioned in section 4.4.5. The data for this initialization is obtained from the table SPSTAB of TABLESF.

OPRTAB: Picks OPRTABi (for pass i) as the initial setting of SPSTAB. This permits a "growing" linkage: in each pass, the code number of an IL operator will be the same as the code number associated with an IL operator with the same name at any other pass.

ONDISK: For special purposes, the user might provide his own initialization of SPSTAB. With this parameter, SPSTAB will be initialized, in pass i, with the contents of the file <ℓ.n.>/SPSTABi which is expected to be on disk.

Default: SPSTAB TERMINALS

6. USE OF ISL - AN EXAMPLE

This chapter is devoted to the use of ISL, i.e., how to describe the semantics of a programming language in ISL. Some general guidelines are initially presented but the most important part is the analysis of an example.

6.1 General Considerations

Obviously, no fixed rules may be given about how to use ISL to describe the semantics of a language; ISL is only a tool and the ingenuity of the programmer plays a very important part in determing how complete and efficient a semantic description is. Some guidelines, however, are presented in this section.

The task of describing the semantics of a language for the ILLIAC IV TWS may be divided into four parts:

a) divide the Semantics into Passes: As explained in Section 2.1.2, it is left to the user to decide how many passes will be employed in the implementation of a given language. It is also quite important to decide how much will be done in each pass, trying to avoid "crowding" one pass. Section 2.1.2 also presented guidelines about what should be done in each pass. Finally, if more than one pass is used, the repertoire of IL operators for each pass must be selected.

b) plan the structure of each pass: In this part the user should decide which tables, stacks, etc. are needed in each pass. The column structure of the tables should be defined, keeping in mind which pointers are needed and how many bits should be reserved for each of them. In pass 1, it is also necessary to decide what use will be made of the SEM fields of BIGTAB and MSTACK. For ALGOL-like languages it is classical to have, in

pass 1, a table of declared identifiers linked with forward and backward pointers to BIGTAB. It is also very common to have a stack to keep track of the block structure (levels) of the ALGOL-like program. Finally, one should decide in this part which information must be transferred between passes and plan the structure of the HEAD.

c) assign semantic calls to the syntax: Usually the syntax is debugged first and in this process the user may discover the need for some semantic tests. These are then the first semantic calls attached to the syntax and may indicate the need to keep some table or stack to provide the information required to "help" the parser. Then the semantic action calls are attached to the syntax according to the structure chosen for the first pass. The syntactic preprocessor then should be used again to make sure that the addition of some semantic call did not make the syntax unacceptable by the Floyd Productions Generator. On assigning semantic calls to the syntax, one should keep in mind the rules given in Section 2.3.2.1 about the situation of MSTACK at each point in the parsing. The purpose of the semantic calls is to transfer control to the semantic routines when tables must be updated and/or IL code has to be emitted. However, it should be noted that, in general, it is better to postpone a semantic action call as much as possible; i.e., insert a semantic action call only when the next reduction would cause the loss of necessary information. In this way the user may have only a few calls and do a lot of things in each of them, rather than having many simple actions which would cause frequent interruptions in the parser.

d) Write the semantic actions for each pass: Once the previous three parts have been taken care of, this should be a relatively straightforward task. The only care would be to use ISL as efficiently as possible.

6.2 Analysis of the Semantic Description of DEMALGOL I

DEMALGOL I is an extremely simple subset of ALGOL which has been implemented with the ILLIAC IV TWS as an example. DEMALGOL I has no arrays, procedures or repetitive statements. There are only two types of quantities: integer and boolean. All labels must be declared in the innermost block in which they occur. There are only three basic statements: assignments, go to's and if's. IF statements must be complete (i.e., there must always be an ELSE part) and may be nested. Blocks may also be nested as usual in ALGOL and the meaning of "scope" of a declaration is retained.

The syntax of DEMALGOL I has been written in TWINKLE in two different ways. Appendix B contains the syntax of DEMALGOL I in BNF TWINKLE productions only. Appendix C contains the same syntax but making use of the English-like constructs available in TWINKLE. Normally, references will be made to Appendix B; Appendix C will be used as an example of how to attach Syntax to the English-like constructs of TWINKLE. Each of the four basic parts pointed out in Section 6.1 is now discussed for the semantic description of DEMALGOL I.

6.2.1 Dividing the Semantics into Passes

Since DEMALGOL is a very simple language, one cannot see, a priori, any necessity for optimization passes. Thus, DEMALGOL will be implemented using the basic number of passes in the system, i.e., 2 passes. The next decision must be the repertoire of IL operators to be used in pass 2. Following the guideline of dealing with the language in pass 1 and with the machine in pass 2, the set of IL operators listed in Table 5 was selected. These operators

seem to be general enough to be machine independent and should enable the generation of efficient machine code for almost any simple computer. Note, however, that all the complexities of the language were dealt with in pass 1: block structure, nesting of if statements, linked lists of forward references, etc. The remainder of the example contains only the implementation of pass 1 which is obviously the most complex in this case. The IL operators have been chosen in such a way that pass 2 can be quite trivial for a 0 address machine (stack machine) and still reasonably simple for a 1 address machine. There are some considerations about a possible implementation of pass 2 for a stack machine: this implementation is so simple that the only information that must be passed between passes is in the IL string; therefore, HEAD may be empty except perhaps for the integer MAXPTABTAB which contains the maximum number of work area locations needed in pass 2. Although not essential, this information might make pass 2 more efficient by allocating dynamically the size of the work area according to the value of MAXPTABTAB. This work area is the only non-simple data structure in pass 2: It might easily be implemented with a stack, and the contents will be absolute addresses as described in Table 5. Most IL operands are pointers to locations of the work area and only the entry field of IL operands is used since there was no need for more than one table (in fact, stack) in pass 2. For a stack machine, the operators: STORE, +, -, $\times$, /, <, >, $\geq$, $\leq$, =, $\neq$, AND, OR are all trivially implemented since the stack automatically stores partial results. GETSPACE and FORGETSPACE are the storage allocation routines. It is also assumed for simplicity that the code generated is kept in core, disk or other storage with random access so that the instructions SETADDRUTRANSF and SETADDRCTRANSF have access to any transfer instruction previously generated in order to set the addresses of forward references.

TABLE 5

IL OPERATORS FOR DEMALGOL I

OPERATOR NAME	pass 2 MSTACK BEFORE	pass 2 MSTACK AFTER	DESCRIPTION
GETSPACE	...D,C,B,A|	...D|	Assign memory space to a quantity of type B (2 = boolean; 3 = integer), initialize it to A and fill the location pointed to by C with the absolute address of the space assigned.
FORGETSPACE	...C,B,A|	...C|	Return the space whose absolute address is in a location pointed to by B and which was utilized by a quantity of type A.
STORE	...D,C,B,A|	...D|	Emit code to Store the quantity whose address is pointed to by C into the location whose address is pointed to by B; the quantity is of type A.
+	...C,B,A|	...C,B'|	Emit code to Add the quantities whose addresses are pointed to by B and A; push in MSTACK a pointer to the address of the result: B'.
-	...C,B,A|	...C,B'|	Emit code to Subtract the quantity whose address is pointed to by A from the quantity whose address is pointed to by B; push in MSTACK a pointer to the address of the result: B'.
×	...C,B,A|	...C,B'|	as above but Multiply.
/	...C,B,A|	...C,B'|	as above but Divide.
AND	...C,B,A|	...C,B'|	as above but And.
OR	...C,B,A|	...C,B'|	as above but Or.
>	...C,B,A|	...C,B'|	Emit code to Compare the quantity whose address is pointed to by B with the quantity whose address is pointed to by A for greater than. Push in MSTACK a pointer to the address of the result: B'.

TABLE 5

(CONTINUED)

OPERATOR NAME	pass 2 MSTACK BEFORE	pass 2 MSTACK AFTER	DESCRIPTION
<	...C,B,A|	...C,B'|	as above but for less than.
≤	...C,B,A|	...C,B'|	as above but for less than or equal to.
≥	...C,B,A|	...C,B'|	as above but for greater than or equal to.
=	...C,B,A|	...C,B'|	as above but for equal to.
≠	...C,B,A|	...C,B'|	as above but for not equal to.
UTRANSF	...B,A|	...B|	Emit an unconditional transfer to the address pointed to by A.
CTRANSF	...C,B,A|	...C|	Emit a conditional transfer on false to the address pointed to by A on the condition whose address is pointed to by B.
SETADDRUTRANSF	...C,B,A|	...C|	Set the address field of the unconditional transfer which is at the address pointed to by A with the address pointed to by B.
SETADDRCTRANSF	...C,B,A|	...C|	As above but conditional transfer.
STOREADDR	...B,A|	...B|	Store in the location pointed to by A the address at which the next instruction will be placed.
ENDOFIT	...A|	...A|	Emit a HALT instruction.

6.2.2 Planning the Structure of a Pass

This section describes the structure of pass 1 semantics for DEMALGOL I. There are four main data structures (besides, of course, BIGTAB and MSTACK). They are the tables IDTAB, LKLIST and BLOCKSTACK and the stack IFSTACK. The structure of the tables is detailed in Figure 15 and the stack is displayed in Figure 16. The SEM field of BIGTAB and MSTACK was redefined using $ADDCOL declarations as illustrated in Figure 17.

Appendix D presents a listing of the pass 1 semantic description of DEMALGOL I in ISL. References will be made to the number of each card, on the right hand side of the page. Lines 6 through 24 contain the declarations of the tables and stack and the $ADDCOL declaration. These declarations should not present any difficulty since they were used as examples in Chapter 4. Seven integers are used in the program besides MAXPTABTAB which has already been discussed. TEMP1, TEMP2 and TEMP3 are used for temporary storage; I is used as index of FOR loops; STORETYPE stores the last type found, i.e., integer, boolean or label. These types were assigned the numbers 3, 2 and 1 respectively and the DEFINE in line 5 allows the use of the mnemonic identifiers INT, BOOL and LAB instead of the numbers above. CULEVEL stores the current level of the program; i.e., the depth of block nesting. It is incremented each time a #BEGIN is found and decremented each time an #END is found. Finally, PTABTAB is a pointer to the next available work area location in pass 2. Note the #UPPERN declaration in line 25 which shows that this was just a syntax check run. The declarations are completed by a FORMAT declaration and a procedure declaration: ERROR(ERRORNUMBER,PTTOBIGTAB). This procedure is called each time the semantics identifies a semantic error. It prints an error message on the file LINE (i.e., the same file the parser will be using to output a listing) containing an error number and the identifier which caused that error. The meaning of each error number (1 through 5) is listed as a comment in cards 31-39.

It is interesting to note how the procedure picks the identifier out of BIGTAB: The parameter is a pointer to the heading of the entry of the identifier in BIGTAB. The $OPD in line 40 picks the number i in column WORDS of BIGTAB and this number is used as the upper limit of a FOR loop which picks the i+1 words following the heading and prints the rightmost six characters of each. Notice also that ERROR increments the global variable NUMERRS as described in Section 2.3.2.3.

To complete the description of the structure of pass 1, it is necessary to describe how the tables and stacks are used. The main table is IDTAB which keeps one entry (i.e., one row) per declaration of each identifier. The field PTTOBIGTAB of each entry in IDTAB points to the heading of the entry of the identifier in BIGTAB. All entries in IDTAB which correspond to the same identifier are linked together by back pointers which are kept in the column BACKPT. If an identifier has BACKPT = 0, this means that it was the first entry of that identifier in IDTAB. Notice that the same identifier can be entered several times in IDTAB because the same identifier can be repeatedly declared in different blocks.

The field PTTOIDTAB of BIGTAB contains a pointer to the latest entry of the identifier in IDTAB which is still useful, i.e., PTTOIDTAB will point to the valid declaration of an identifier at a certain level while the chain of BACKPT's links entries of the same identifier in outer blocks (i.e., lower levels). The remaining fields of IDTAB are used as follows: PTTOADTAB keeps a pointer to the work area location of pass 2 that will contain the absolute address associated with that entry. LEVEL stores the value of CULEVEL at the moment that entry was made. TYPE designates which type the entry correspond to: BOOL, INT or LAB. The three remaining fields are used only for labels: USED is a bit that is set when a declared label appears in a GOTO statement; FOUND is set when a declared label is found labeling a statement and PTTOLKLIST

points to the latest of the entries in a linked list of forward references to a label. This linked list is kept in the table LKLIST. Column LINK of this table points to the previous entry in the linked list. LINK = 0 means the initial (i.e., the oldest) entry in the list. The other field of LINKLIST : PTTOABTAB contains a pointer to the work area location in pass 2 which contains the address of the transfer instruction whose destination address must be set when the forward label is finally found. Table BLOCKSTACK is used as a stack to recover pass 2 work area space (field PTABTAB) and linked list space (field PTLKLIST). When a block is entered, the values of PTABTAB and PTLKLIST are pushed into BLOCKSTACK. When the block is left, BLOCKSTACK is popped into these two locations, thus recovering the locations that are no longer needed since the block ended. The stack IFSTACK allows the implementation of nested IF statements. Obviously, each IF statement in DEMALGOL needs two transfer instructions: one conditional and one unconditional and both are to forward points. Thus, the address of these transfers must be stored so that the address field may be fixed when the desired point is found. IFSTACK keeps, before the ELSE is found, a pointer to the address of the conditional transfer and, after the ELSE appeared but before the end of the IF statement is found, a pointer to the address of the unconditional transfer. This completes the description of the basic structure of pass 1. Further details are presented in the analysis of the semantic actions.

6.2.3 Assigning Semantic Calls to the Syntax and Writing the Semantic Actions

The assignment of semantic calls is analyzed in detail for the BNF syntax of DEMALGOL (Appendix B). As each semantic call is discussed, the corresponding semantic action is described. This implementation of DEMALGOL pass 1 uses only 21 semantic actions directly called and one auxiliary action (action "22") which is called from actions "7" and "18" by means of the $EXEC construct.

The first production (I-line 1) defines a program and action "1" has been assigned to it simply to emit the IL operator ENDOFIT (D-line 50). ENDOFIT generates code for a HALT instruction, or a call to SYSTEM in more elaborate installations.

Semantic action "2" is called when an #END is found (B-line 2). This semantic routine must "erase" all the declarations which were found in that block. This is done as follows (D-lines 51-62): starting with the last occupied position of IDTAB all declarations with LEVEL = CULEVEL are erased by setting PTTOIDTAB in BIGTAB to the previous declaration (which now is valid again) of the same identifier. This previous declaration is located using BACKPT.

As integer or boolean declarations are erased, the appropriate FORGETSPACE operators are emitted to recover the memory locations which were assigned to these variables. As label declarations are erased, a check is performed to find out if a forward reference was never satisfied; i.e., if a label was used in a GOTO statement but never found. This situation defines semantic error #2. The "erasure" proceeds until an entry is found with LEVEL < CULEVEL indicating that all the declaration part of the block that ended was taken care of. Then the value of MAXPTABTAB is updated to the maximum between the previous MAXPTABTAB and the value reached in this block. BLOCKSTACK is popped (D-line 64) as already described and CULEVEL is decremented.

When a #BEGIN is found, semantic action "3" is called. This action simply increments CULEVEL and pushes BLOCKSTACK (D-lines 66-67). Actions "4", "5" and "6" are called when each type is found (B-line 8) and simply set STORETYPE to the appropriate value. Action "7", called when each identifier is found in a <type list> (B-line 9) performs the entry in IDTAB. PTTOIDTAB of BIGTAB is adjusted to point to this latest entry and BACKPT is set. Note,

however, that if PTTOBIGTAB points to an entry with LEVEL = CULEVEL, this means that the identifier has already been declared before in the current block. Thus, semantic error #1 is detected. Note also that, if the declaration was for an integer or a boolean, the operator GETSPACE is emitted preceded by the appropriate parameters. One location of pass 2 work area (PTABTAB) is reserved for each entry in IDTAB. Auxiliary action "22" performs the actual entry in IDTAB.

Action "8" is called when a label is found (B-line 16) and Action "9" when a label is used (B-line 17). When the label is found, a check is performed to make sure not only that it was declared (semantic error #4) but also that the declaration occurred in the innermost block (semantic error #3). Action "8" also emits code to fill the pass 2 work area location associated with the label with the actual absolute address to which the label corresponds (D-lines 85-86). Finally the linked list of forward references, if any, is eliminated with the emission of one operator SETADDRUTRANSF (preceded by the appropriate parameters) per forward reference (D-lines 90-92). Action "9" must output a UTRANSF operator and, if the label has not been found yet, enter a pointer to the address of the unconditional transfer in the linked list associated with that label.

Action "10" is the only semantic test needed in this program. Its reason and effect have already been discussed since this case was used as an example in Section 2.3.2.2. The semantic test is needed twice in this language: B-line 20 and B-line 44.

Actions "11" and "12" emit STORE operators to implement the assignment states: arithmetic and boolean. Note that Action "12" also checks for the previous declaration of the left hand side identifier as a boolean (D-line 117) and this constitutes the last semantic error: #5. Action "11" does not have to perform this kind of check because it has already been performed by "10" in the semantic test.

Actions "13", "14" and "15" implement IF statements. "13" is called after the boolean expression is found and basically emits a CTRANSF. "14" is called when the THEN part is complete. It fixes the conditional transfer and emits UTRANSF which is fixed by "15", when the ELSE part is complete. The use of IFSTACK to make possible the implementation of nested IF's has already been discussed.

Action "16" is particularly interesting: It is the only use of a dynamic operator in this program. It is used in five different calls (B-lines 38, 42, 48, 50 and 54) and is solely responsible for the emission of the operators: +, -, ×, /, AND, OR, >, ≥, <, ≤, =, and ≠. Obviously, this language will use the option SPSTAB TERMINALS in pass 1 to create the link between the terminals code numbers and IL operators code numbers. Note that postfix code results automatically.

Actions "17" and "19" emit IL operands corresponding to identifiers which have been parsed as arithmetic primaries and boolean primaries respectively. As in "11" and "12", "19" checks for error #5 while "17" does not since "10", called just before "17," has already done the checking.

Action "18" takes care of numbers. Numbers are treated as self-declarations and an entry in IDTAB is made for each number the first time it appears in a block. When the bit NUMBEROK of BIGTAB is zero, it means that the number has not yet appeared in this block and thus it must be created. The initialization parameter of the GETSPACE operator is used to give the location reserved for the number the proper initial value.

Note that #TRUE and #FALSE are not considered self-declarations as are the numbers. In fact, they are treated (in Actions "20" and "21") with references to fixed pass 2 work areas: locations 0 and 1 which, in the initialization action "0", were assigned to point to the absolute address of the constants #FALSE and #TRUE respectively. Thus, instead of being created once for

each block in which they appear, #TRUE and #FALSE are created only once, at the initialization, as if they were the two first declarations in the outermost block.

This semantic description of DEMALGOL I has actually been used to create a compiler and to parse sample programs. The listings corresponding to these tests can be found in [4].

The assignment of semantic action calls to the English-like version of the syntax (Appendix C) is a little more complex. The difficulty, however, is simply solved if one compares Appendices B and C to determine where a semantic call should be placed in the English-like form to obtain the same effect as in the BNF form. Very often brackets must be used when associating semantics to the English-like constructs of TWINKLE (See, for example, C-line 10).

APPENDIX A

FORMAL SYNTAX OF ISL

This appendix contains the formal definition of ISL. This definition is presented through BNF productions. There is, however, one major explanation: Since ISL contains BeA, it would be rather extensive to present all of it here. Therefore, only the non-ALGOL ISL constructs are defined here. The definitions of the BeA constructs given in [1] are in effect and they are referred to by the following notation "BeA n-m:", where n and m are integers. Whenever this notation appears inside non-terminal brackets before the non-terminal name, it indicates that the definition of the non-terminal is given in [1], page n-m. In order to use [1] as a complement of the definition of ISL, however, certain definitions in this BeA manual should be added to in order to include ISL constructs. This is what is done in productions 1-7 below. These are the only productions in which a BeA reference appears in the left-hand side; moreover, the same reference appears in the right-hand side. The meaning of these productions is: the referred BeA non-terminal should be modified to include, besides its original definition, the indicated ISL constructs. Finally, many of the non-terminals which appear in the remaining of the syntax are qualified by footnotes indicating restrictions which cannot be conveniently expressed in BNF.

1. <BeA 9-1 : declaration> ::= <BeA 9-1 : declaration> | <ISL declaration>

2. <BeA 4-14 : label> ::= <BeA 4-14 : label> | <action label>

3. <BeA 6-2 : unlabeled basic statement> ::=
<BeA 6-2 : unlabeled basic statement> | <ISL simple statement>

4. <BeA 5-1 : unlabeled compound statement> ::=
<BeA 5-1 : unlabeled compound statement> | <ISL compound statement>

5. <BeA 3-3 : partial word designator> ::=
<BeA 3-3 : partial word designator> | <ISL operand>

6. <BeA 3-7 : function designator> ::=
<BeA 3-7 : function designator> | <search operand>

7. <BeA 3-1 : simple variable> ::= <BeA 3-1 : simple variable> |
<pointer operand>

8. <program> ::= $BEGIN <declaration part> ; <action label> :
<executable part> $END | $BEGIN <action label> :
<executable part> $END

9. <declaration part> ::= <list of declarations>

10. <list of declarations> ::= <BeA 9-1 : declaration> |
<list of declarations> ; <BeA 9-1 : declaration>

11. <action label> ::= $ACTION (<generalized identifier>[1])

12. <generalized identifier> ::= <BeA 2-1 : letter> | <BeA 2-1 : digit> |
<generalized identifier> <BeA 2-1 : letter> |
<generalized identifier> <BeA 2-1 : digit>

13. <executable part> ::= <BeA 1-4 : empty> |
<executable part> ; <BeA 6-1 : statement>

14. <ISL declaration> ::= <table declaration> | <stack declaration> |
<cell declaration> | <add column declaration> |
<number of actions declaration> | <in declaration> |
< ISL define declaration> |
<forward action declaration> |
<end of declarations declaration>

15. <ISL simple statement> ::= <increment statement> |
<decrement statement> |
<execute statement>

16. <ISL compound statement> ::= <push statement> | <pop statement>
<enter statement> | <code statement>

17. <table declaration> ::= $TABLE (<list of table specifications>)

18. <list of table specifications> ::= <table specification > |
<list of table specifications> ;
<table specification>

19. <table specification> ::= <table identifier> , <number of rows> ,
<list of column specifications>

20. <list of column specifications> ::= <column specification> |
<list of column specifications> ,
<column specification>

21. <column specification> ::= <column identifier> = <left bit of field> :
< bits in field>

22. <number of rows> ::= <BeA 2-6 : unsigned integer>

23. <left bit of field> ::= <BeA 2-6 : unsigned integer>

24. <bits in field> ::= <BeA 2-6 : unsigned integer>

25. <table identifier> ::= <BeA 2-5 : identifier^{2}>

26. <column identifier> ::= <BeA 2-5 : identifier^{3}>

27. <stack declaration> ::= $STACK (<list of stack specifications>)

28. <list of stack specifications> ::= <stack specification> |
<list of stack specifications> ;
<stack specification>

29. <stack specification> ::= <stack identifier> , <number of rows>

30. <stack identifier> ::= <BeA 2-5 : identifier^{2}>

31. <cell declaration> ::= $CELL (<list of cell specifications>)

32. <list of cell specifications> ::= <cell specification> |
<list of cell specifications> ;
<cell specification>

33. <cell specification> ::= <cell identifier> , <list of column specifications>

34. <cell identifier> ::= <BeA 2-5 : identifier^{2}>

35. <add column declaration> ::= $ADDCOL (<list of add column specifications>)

36. <list of add column specifications> ::= <add column specification> |
<list of add column specifications>
<add column specification>

37. <add column specification> ::= <table identifier[4]> ,
<list of column specifications> |
<cell identifier[5]> ,
<list of column specifications>

38. <number of actions declaration> ::= $UPPERN (<BeA 2-6 : unsigned integer>)

39. <in declaration> ::= $IN (<list of declarations[6]>)

40. <ISL define declaration> ::= $DEFINE (<ISL definition list>)

41. <ISL definition list> ::= <ISL definition part> |
<ISL definition list> , <ISL definition part>

42. <ISL definition part> ::= <ISL defined identifier> =
<BeA 9-7 : definition> #

43. <ISL defined identifier> ::= <BeA 2-5 : identifier[2]>

44. <forward action declaration> ::= $FORWARD (<list of action identifiers>)

45. <list of action identifiers> ::= <generalized identifier[7]> |
<list of action identifiers> ,
<generalized identifier[7]>

46. <end of declarations declaration> ::= $ENDEC[8]

47. <ISL operand> ::= <table operand> | <stack operand> | <cell operand>

48. <table operand> ::= $OPD (<table identifier[4]>) , <row information> ,
<column information>)

49. <stack operand> ::= $OPD (<stack identifier[9]> , <row information>)

50. <cell operand> ::= $OPD (<cell identifier[5]> , <column information>)

51. <row information> ::= <BeA 4-1 : arithmetic expression> | $NOW | $NEXT
$TOP | $TOP <BeA 4-2 : adding operator>
<BeA 4-1 : arithmetic expression>

52. <column information> ::= <column identifier[10]> | $ALL[11] |
$ALL <BeA 4-1 : arithmetic expression>[12]

53. <search operand> ::= <stack search operand> | <table search operand>

54. <stack search operand> ::= $SEARCH (<stack identifier[9]> ,
<BeA 4-1 : arithmetic expression>)

55. <table search operand> ::= $SEARCH (<table identifier[4]> ,
<column information> ,
<BeA 4-1 : arithmetic expression>)

56. <pointer operand> ::= $POINTER (<table identifier[4]>) |
$POINTER (<stack identifier[9]>)

57. <increment statement> ::= $INCR (<leftmost left part>)

58. <leftmost left part> ::= <BeA 3-3 : partial word designator> |
<BeA 3-1 : variable>

59. <decrement statement> ::= $DECR (<leftmost left part>)

60. <execute statement> ::= $EXEC (<generalized identifier[1]>)

61. <push statement> ::= <stack push statement> | <table push statement>

62. <stack push statement> ::= $PUSH (<stack identifier^{9}> :
<BeA 4-1 : arithmetic expression>)

63. <table push statement> ::= $PUSH (<table identifier^{4}> , <entry list>)

64. <entry list> ::= <entry element> | <entry list> , <entry element>

65. <entry element> ::= <column information> : <BeA 4-1 : arithmetic expression>

66. <pop statement> ::= <stack pop statement> | <table pop statement>

67. <stack pop statement> ::= $POP (<stack identifier^{9}> :
<leftmost left part>) |
$POP (<stack identifier^{9}>)

68. <table pop statement> ::= $POP (<table identifier^{4}> , <pop entry list>) |
$POP (<table identifier^{4}>)

69. <pop entry list> ::= <pop entry element>
<pop entry list> , <pop entry element>

70. <pop entryelement> ::= < column information> : <leftmost left part>

71. <enter statement> ::= <cell enter statement> | <table enter statement>

72. <cell enter statement> ::= $ENTER (<cell identifier^{5}> , <entry list>)

73. <table enter statement> ::= $ENTER (<table identifier^{4}> ,
<row information> , entry list>)

74. <code statement> ::= $CODE (<list of operators and operands>)

75. <list of operators and operands> ::= <operator> | <operand> |
<list of operators and operands> ,
<operand> |
<list of operators and operands> ,
<operator>

76. <operand> ::= <BeA 4-1 : arithmetic expression>

77. <operator> ::= <dynamic operaotr> | <static operator>

78. <dynamic operator> ::= #<BeA 4-1 ' arithmetic expression>

79. <static operator> ::= *"<BeA 1-4 : character>" |
* <generalized identifier>

80. <definition call> ::= $DF <ISL defined identifier[13]>[14]

FOOTNOTES:

1. The generalized identifier must have appeared before in a <forward action declaration> or in a semantic action call or semantic test in the syntax. Moreover, this construct can label only statements in the outermost block.

2. The identifier must not have been used before in another declaration in the same block.

3. The identifier must not have been used before in the same <table specification>.

4. The table identifier must have appeared previously in a <table declaration> now in effect.

5. The cell identifier must have appeared previously in a <cell declaration> now in effect.

6. The list of declarations may not contain an <in declaration>.

7. The generalized identifier must later appear in an <action label>.

8. This declaration, when present, must be the last declaration in the block.

9. The stack identifier must have appeared previously in a <stack declaration> now in effect.

10. The column identifier must have appeared previously in the <table declaration or <cell declaration> referred to by the construct.

11. This type of column information myst be used with tables or cells that do not use more than one word per row.

12. This type of column information must be used only with tables or cells that use more than one word per row.

13. The ISL defined identifier must have appeared previously in an <ISL define declaration> now in effect. Moreover, there must be no space between the $DF and the identifier.

14. Definition calls may appear at any place in which the corresponding <definition> might appear.

APPENDIX B

SYNTAX OF DEMALGOL I IN TWINKLE - BNF FORM

```
<PROGRAM> ::= <BLOCK> @S 1 ;                                                    1
<BLOCK> ::= <HEAD> <TAIL> #END @S 2 ;                                           2
<HEAD> ::= #BEGIN @S 3 <DECLARATION PART> #; / #BEGIN @S 3 ;                     3
<DECLARATION PART> ::= <DECLARATION PART> <SEPARATOR>                           4
                       <TYPE> <TYPE LIST> ;                                     5
<DECLARATION PART> ::= <TYPE> <TYPE LIST> ;                                     6
<SEPARATOR> ::= #; / <COMMENT> ;                                                7
<TYPE> ::= #INTEGER @S 4 / #LABEL @S 5 / #BOOLEAN @S 6 ;                        8
<TYPE LIST> ::= <TYPE LIST> #, <*I> @S 7 / <*I> @S 7 ;                          9
<TAIL> ::= <TAIL> <SEPARATOR> <STATEMENT> / <STATEMENT> ;                       10
<STATEMENT> ::= <BLOCK> /                                                       11
                <GO TO STATEMENT> /                                             12
                <ASSIGNMENT> /                                                  13
                <IF PART> <THEN PART> <ELSE PART> /                             14
                < > /                                                           15
                <LABEL> @S 8 #: <STATEMENT> ;                                   16
<GO TO STATEMENT> ::= <GO TO SYMBOL> <LABEL> @S 9 ;                             17
<GO TO SYMBOL> ::= #GOTO / #GO #TO / #GO ;                                      18
<LABEL> ::= <*I> ;                                                              19
<ASSIGNMENT> ::= <*I> @T 10 <ASSIGN ARROW>                                      20
                   <ARITHMETIC EXPRESSION> @S 11 /                              21
                 <*I> <ASSIGN ARROW> <BOOLEAN EXPRESSION> @S 12 ;               22
<ASSIGN ARROW> ::= #← / #: #= ;                                                 23
<IF PART> ::= #IF <BOOLEAN EXPRESSION> @S 13 ;                                  24
<THEN PART> ::= #THEN <STATEMENT> @S 14 ;                                       25
<ELSE PART> ::= #ELSE <STATEMENT> @S 15 ;                                       26
<COMMENT> ::= #COMMENT <COMMENT BODY> #; ;                                      27
<COMMENT BODY> ::= <ANYTHING BUT SEMICOLON> /                                   28
```

```
                <COMMENT BODY> <ANYTHING BUT SEMICOLON> ;                    29
NYTHING BUT SEMICOLON> ::= <*I> / <*N> / <*S> / #BEGIN / #END /              30
                           #GO / #TO / #GOTO / #COMMENT /                    31
                           #INTEGER / #BOOLEAN / #LABEL / #IF /              32
                           #THEN / #ELSE / #AND / #OR /                      33
                           #TRUE / #FALSE /                                  34
                           #, / #: / #= / #@ / #. ;                          35
RITHMETIC EXPRESSION> ::= <TERM> /                                           36
                          <ARITHMETIC EXPRESSION>                            37
                            <ADDING OPERATOR> <TERM> @S 16 ;                 38
DDING OPERATOR> ::= #+ / #- ;                                                39
RM> ::= <ARITHMETIC PRIMARY> /                                               40
        <TERM> <MULTIPLYING OPERATOR>                                        41
          <ARITHMETIC PRIMARY> @S 16 ;                                       42
ULTIPLYING OPERATOR> ::= #x / #/ ;                                           43
RITHMETIC PRIMARY> ::= <*I> @T 10 @S 17 / <*N> @S 18 /                       44
                       #( <ARITHMETIC EXPRESSION> #) ;                       45
OOLEAN EXPRESSION> ::= <BOOLEAN TERM> /                                      46
                       <BOOLEAN EXPRESSION> #OR                              47
                         <BOOLEAN TERM> @S 16 ;                              48
OOLEAN TERM> ::= <BOOLEAN PRIMARY> /                                         49
                 <BOOLEAN TERM> #AND <BOOLEAN PRIMARY> @S 16 ;               50
OOLEAN PRIMARY> ::= <*I> @S 19 / #TRUE @S 20 / #FALSE @S 21 /                51
                    #( <BOOLEAN EXPRESSION> #) /                             52
                    <ARITHMETIC EXPRESSION> <RELATIONAL OPERATOR>            53
                      <ARITHMETIC EXPRESSION> @S 16 ;                        54
LATIONAL OPERATOR> ::= #= / ## / #< / #<= / #> / #>= ;                        55
```

APPENDIX C

SYNTAX OF DEMALGOL I IN TWINKLE - ENGLISH-LIKE FORM

```
A <PROGRAM> IS DEFINED TO BE A <BLOCK> #S 1 ;                                1
                                                                             2
A <BLOCK> CONSISTS OF #BEGIN #S3 FOLLOWED BY POSSIBLY ONE                    3
[LIST OF <DECLARATION>S SEPARATED BY <COMMENT>S FOLLOWED BY #; ]             4
FOLLOWED BY A LIST OF <STATEMENT>S SEPARATED BY <COMMENT>S                   5
FOLLOWED BY #END #S2 ;                                                       6
                                                                             7
A <DECLARATION> CONSISTS OF                                                  8
#INTEGER #S4 / #BOOLEAN #S5 / #LABEL #S6 ]                                   9
FOLLOWED BY A STRING OF [<*I> #S7 ] SEPARATED BY #, ;                        10
                                                                             11
A <STATEMENT> IS DEFINED TO BE [ <LABEL> #S8 #: ]* FOLLOWED BY               12
[ [ #GO #TO OR #GO OR #GOTO ] FOLLOWED BY A <LABEL> #S9 OR                   13
<*I> #T10 [#: #= OR #← ] <ARITHMETIC EXPRESSION> #S11 OR                     14
<*I> [ #: #= OR #← ] <BOOLEAN EXPRESSION> #S12 OR                            15
#IF <BOOLEAN EXPRESSION> #S13 #THEN <STATEMENT> #S14 #ELSE <STATEMENT>       16
#S15 OR EMPTY ] ;                                                            17
                                                                             18
A <COMMENT> CONSISTS OF #; OR #COMMENT FOLLOWED BY A POSSIBLY                19
EMPTY LIST OF <ANYTHING BUT SEMICOLON>S FOLLOWED BY #; ;                     20
                                                                             21
A <LABEL> IS DEFINED TO BE AN <*I> (PROVIDED IT HAS APPEARED                 22
IN A DECLARATION BEGINNING WITH #LABEL));                                    23
                                                                             24
 N <ARITHMETIC EXPRESSION> CONSISTS OF A <TERM> FOLLOWED BY A POSSIBLY       25
EMPTY LIST OF [[ #+ OR #- ] FOLLOWED BY A <TERM> #S16 ] ;                    26
                                                                             27
A <TERM> CONSISTS OF AN <ARITHMETIC PRIMARY> FOLLOWED BY A POSSIBLY          28
```

```
EMPTY LIST OF [[ #× OR #/ ] FOLLOWED BY AN <ARITHMETIC PRIMARY> @S16]}                29
                                                                                     30
AN <ARITHMETIC PRIMARY> CONSISTS OF #( <ARITHMETIC EXPRESSION> #) OR                 31
AN <*I>@T10 @S17 OR A <*N> @S18 }                                                    32
                                                                                     33
A <BOOLEAN EXPRESSION> CONSISTS OF A <BOOLEAN TERM> FOLLOWED BY A                    34
POSSIBLY EMPTY                                                                       35
LIST OF [ #OR FOLLOWED BY A <BOOLEAN TERM> @S16 ]}                                   36
                                                                                     37
A <BOOLEAN TERM> IS DEFINED TO BE A <BOOLEAN PRIMARY> FOLLOWED BY A                  38
POSSIBLY EMPTY LIST OF [ #AND <BOOLEAN PRIMARY> @S16 ] }                             39
                                                                                     40
A <BOOLEAN PRIMARY> CONSISTS OF #( <BOOLEAN EXPRESSION> #) OR                        41
AN <ARITHMETIC EXPRESSION> FOLLOWED BY [ #= / #≠ / #> / #< / #≤ / #≥]                42
FOLLOWED BY AN <ARITHMETIC EXPRESSION> OR AN <*I> @S19 OR                            43
#TRUE @S20 OR #FALSE @S21 }                                                          44
                                                                                     45
AN <ANYTHING BUT SEMICOLON> CONSISTS OF <*I> / <*N> / <*S> / #BEGIN /                46
#END / #GO / #TO / # GOTO / #COMMENT / #INTEGER / #BOOLEAN / #LABEL /                47
#IF / #THEN / #ELSE / #AND / #OR / #TRUE /#FALSE /#, / #1 / #2 / #0 /                48
#. }                                                                                 49
```

APPENDIX D

SEMANTICS OF DEMALGOL I IN ISL - PASS 1

```
$#                    SEG 100+100 NEWSEQ USEIL                                  1
 $BEGIN                                                                         2
    INTEGER STORETYPE, CULEVEL, PTABTAB, TEMP1, TEMP2, TEMP3 ,I ;               3
    INTEGER MAXPTABTAB ;                                                        4
    DEFINE LAB=1#, BOOL=2#, INT=3#, LABELUSEDBUTNOTFOUND=6# ;                   5
    $TABLE(IDTAB,1022,PTTOBIGTAB= 1:13,                                         6
                      BACKPT     =14:10,                                        7
                      PTTOABTAB =24:8 ,                                         8
                      LEVEL      =32:4 ,                                        9
                      TYPE       =36:2 ,                                        10
                      USED       =38:1 ,                                        11
                      FOUND      =39:1 ,                                        12
                      LABFIELD   =36:4 ,                                        13
                      PTTOLKLIST=40:8 ,                                         14
                      REST       =38:10 ;                                       15
           LKLIST,256,PTTOABTAB= 1:8 ,                                          16
                      LINK      = 9:8 ;                                         17
           BLOCKSTACK,16,PTABTAB = 1:8 ,                                        18
                        PTLKLIST= 9:8 ) ;                                       19
    $STACK(IFSTACK,32) ;                                                        20
    $ADDCOL(BIGTAB,PTTOIDTAB=6:10 ,                                             21
                   NUMBEROK =5:1  ;                                             22
            MSTACK,PTTOIDTAB=6:10 ,                                             23
                   NUMBEROK =7:1 ) ;                                            24
    $SUPPERN(22) ;                                                              25
    FORMAT ERRORMESSAGE("***** SEMANTIC ERROR #",I1,                            26
                        " , IDENTIFIER IS : ",8A6) ;                            27
    PROCEDURE ERROR(ERRORNUMBER,PTTOBIGTAB);                                    28
      VALUE ERRORNUMBER,PTTOBIGTAB; INTEGER ERRORNUMBER,PTTOBIGTAB;             29
```

```
BEGIN                                                                   30
   COMMENT SEMANTIC ERRORS IN DEMALGOL I :                              31
     1 - ID HAS ALREADY BEEN DECLARED IN THIS BLOCK.                    32
     2 - A LABEL DECLARED AND USED IN A BLOCK DID NOT APPEAR.           33
     3 - LABEL WAS NOT DECLARED IN THE INNERMOST BLOCK IN WHICH         34
         IT OCCURS.                                                     35
     4 - ID HAS NOT BEEN DECLARED AS A LABEL OR HAS NOT BEEN            36
         DECLARED AT ALL.                                               37
     5 - ID HAS NOT BEEN DECLARED AS A BOOLEAN OR HAS NOT BEEN          38
         DECLARED AT ALL. ;                                             39
   TEMP3:=$OPD(BIGTAB,PTTOBIGTAB,WORDS);                                40
   WRITE(LINE,ERRORMESSAGE,ERRORNUMBER,FOR I:=0 STEP 1 UNTIL TEMP3      41
            DO $OPD(BIGTAB,PTTOBIGTAB+I+1,ALL));                        42
   $INCR(NUMERRS)                                                       43
END OF PROCEDURE ERROR ;                                                44
                                                                        45
******** BEGINNING OF THE SEMANTIC ACTIONS **********                   46
                                                                        47
$ACTION( 0): CULEVEL:=0; PTIOTAB:=PTLKLIST:=1; PTABTAB:=2;              48
          $CODE(0,BOOL,0,*GETSPACE,1,BOOL,1,*GETSPACE) ;                49
$ACTION( 1): $CODE(*ENDOFIT);                                           50
$ACTION( 2): FOR I:=PTIDTAB-1 STEP -1 WHILE $OPD(IDTAB,I,LEVEL)=        51
                                                  CULEVEL DO            52
             BEGIN                                                      53
               $OPD(BIGTAB,$OPD(IDTAB,I,PTTOBIGTAB),PTTOIDTAB):=        54
                                    $OPD(IDTAB,I,BACKPT) ;              55
               IF $OPD(IDTAB,I,LABFIELD)=LABELUSEDBUTNOTFOUND           56
                 THEN ERROR(2,$OPD(IDTAB,I,PTTOBIGTAB)) ;               57
               IF TEMP1:=$OPD(IDTAB,I,TYPE)#LAB THEN                    58
                 $CODE($OPD(IDTAB,I,PTTOABTAB),                         59
```

```
                    TEMP1,*FORGETSPACE);                                    60
                PTIDTAB*I                                                   61
              END;                                                          62
            MAXPTABTAB*MAX(MAXPTABTAB,PTABTAB);                             63
            $POP(BLOCKSTACK,PTABTAB:PTABTAB,PTLKLIST:PTLKLIST);             64
            $DECR(CULEVEL);                                                 65
$ACTION( 3): $INCR(CULEVEL);                                                66
            $PUSH(BLOCKSTACK,PTABTAB:PTABTAB,PTLKLIST:PTLKLIST);            67
$ACTION( 4): STORETYPE*INT ;                                                68
$ACTION( 5): STORETYPE*LAB ;                                                69
$ACTION( 6): STORETYPE*BOOL ;                                               70
$ACTION( 7): IF (IF TEMP1*$OPD(MSTACK,NOW,PTTOIDTAB)=0                      71
                  THEN TRUE                                                 72
                  ELSE $OPD(IDTAB,TEMP1,LEVEL)<CULEVEL)                     73
               THEN BEGIN                                                   74
                      $EXEC(22);                                            75
                      IF STORETYPE#LAB THEN                                 76
                        $CODE(PTABTAB,STORETYPE,0,*GETSPACE);               77
                      $INCR(PTABTAB)                                        78
                    END                                                     79
               ELSE ERROR(1,$OPD(MSTACK,NOW,ENTRY));                        80
$ACTION( 8): IF $OPD(IDTAB,TEMP1*$OPD(MSTACK,NOW,PTTOIDTAB),TYPE)=          81
                                                        LAB                 82
               THEN IF $OPD(IDTAB,TEMP1,LEVEL)=CULEVEL                      83
                      THEN BEGIN                                            84
                             $CODE(TEMP3*$OPD(IDTAB,TEMP1,PTTOABTAB)        85
                                   ,*STOREADD);                             86
                             TEMP2*$OPD(IDTAB,TEMP1,PTTOLKLIST);            87
                             WHILE TEMP2#0 DO                               88
                             BEGIN                                          89
                               $CODE(TEMP3,                                 90
```

```
                                $OPD(LKLIST,TEMP2,PTTOABTAB),                    91
                                *SETADDRUTRANSF ));                              92
                            TEMP2←$OPD(LKLIST,TEMP2,LINK)                        93
                          END);                                                  94
                          $OPD(IDTAB,TEMP1,FOUND)←1                              95
                        END                                                      96
                      ELSE ERROR(3,$OPD(MSTACK,NOW,ENTRY))                       97
                ELSE ERROR(4,$OPD(MSTACK,NOW,ENTRY)));                           98
$ACTION( 9): IF $OPD(IDTAB,TEMP1←$OPD(MSTACK,NOW,PTTOIDTAB),TYPE)=               99
                                                        LAB                     100
                THEN BEGIN                                                      101
                       IF $OPD(IDTAB,TEMP1,FOUND)=0 THEN                        102
                         BEGIN                                                  103
                           $PUSH(LKLIST,PTTOABTAB:PTABTAB,                      104
                                 LINK:$OPD(IDTAB,TEMP1,PTTOLKLIST)));           105
                           $ENTER(IDTAB,TEMP1,PTTOLKLIST:PTLKLIST=1,            106
                                              USED:1 ));                        107
                           $CODE(PTABTAB,*STOREADDR)); $INCR(PTABTAB)           108
                         END);                                                  109
                       $CODE($OPD(IDTAB,TEMP1,PTTOABTAB),*UTRANSF)              110
                     END                                                        111
                ELSE ERROR(4,$OPD(MSTACK,NOW,ENTRY)));                          112
$ACTION(10): SEMANTICTEST←$OPD(IDTAB,$OPD(MSTACK,NOW,PTTOIDTAB),                113
                               TYPE) = INT );                                   114
$ACTION(11): $CODE($OPD(IDTAB,$OPD(MSTACK,PTMSTACK-3,PTTOIDTAB),                115
                        PTTOABTAB),INT,*STORE));                                116
$ACTION(12): IF $OPD(IDTAB,TEMP1←$OPD(MSTACK,PTMSTACK-3,PTTOIDTAB),             117
                     TYPE)=BOOL                                                 118
                THEN $CODE($OPD(IDTAB,TEMP1,PTTOABTAB),BOOL,*STORE)             119
                ELSE ERROR(5,$OPD(MSTACK,PTMSTACK-3,ENTRY)));                   120
```

```
$ACTION(13): $PUSH(IFSTACK,PTABTAB);                                    121
             $CODE(PTABTAB,*STOREADDR,0,*CTRANSF);                      122
             $INCR(PTABTAB);                                            123
$ACTION(14): $POP(IFSTACK,TEMP1); TEMP2←PTABTAB+1;                      124
             $CODE(PTABTAB,*STOREADDR,0,*UTRANSF,TEMP2,*STOREADDR,      125
                   TEMP2,TEMP1,*SETADDRCTRANSF);                        126
             $PUSH(IFSTACK,PTABTAB); $INCR(PTABTAB);                    127
$ACTION(15): $CODE(PTABTAB,*STOREADDR,PTABTAB,$OPD(IFSTACK,NOW),        128
                   *SETADDRUTRANSF);                                    129
             $POP(IFSTACK);                                             130
$ACTION(16): $CODE(#$OPD(MSTACK,PTMSTACK-2,ENTRY));                     131
$ACTION(17): $CODE($OPD(IDTAB,$OPD(MSTACK,NOW,PTTOIDTAB),               132
                                          PTTOABTAB));                  133
$ACTION(18): IF TEMP1←$OPD(MSTACK,NOW,NUMBEROK)=0                       134
               THEN BEGIN                                               135
                      $EXEC(22);                                        136
                      $OPD(BIGTAB,TEMP2,NUMBEROK)←1;                    137
                      $CODE(PTABTAB,INT,                                138
                            $OPD(BIGTAB,TEMP2←1,ALL),                   139
                            *GETSPACE,PTABTAB);                         140
                      $INCR(PTABTAB)                                    141
                    END                                                 142
             ELSE $CODE($OPD(IDTAB,$OPD(MSTACK,NOW,PTTOIDTAB),          143
                             PTTOABTAB));                               144
$ACTION(19): IF $OPD(IDTAB,TEMP1←$OPD(MSTACK,NOW,PTTOIDTAB),TYPE)=      145
                                                          BOOL          146
             THEN $CODE($OPD(IDTAB,TEMP1,PTTOABTAB))                    147
             ELSE ERROR(5,$OPD(MSTACK,NOW,ENTRY));                      148
$ACTION(20): $CODE(1);                                                  149
$ACTION(21): $CODE(0);                                                  150
$ACTION(22): $OPD(BIGTAB,TEMP2←$OPD(MSTACK,NOW,ENTRY),PTTOIDTAB)←       151
```

```
                                                    PTIDTAB);
$PUSH(IDTAB,PTTOBIGTAB:TEMP2,
        BACKPT      :TEMP1,
        PTTOABTAB   :PTABTAB,
        LEVEL       :CULEVEL,
        TYPE        :STORETYPE,
        REST        :0 )
```

APPENDIX E

OUTPUT PRODUCED BY THE ISL TRANSLATOR

ON THE SEMANTICS OF DEMALGOL I - PASS 1

```
$#                  SEQ 100+100 NEWSEQ USEIL
 $BEGIN
                                    ++++++++++  $ CARD                                                                  00000100
                                    ++++++++++  $$ C SCANNER                                                            00000200
                                    ++++++++++  $$ C IPARSER                                                            00000300
                                    ++++++++++                                                                          00000400
                                    ++++++++++                                                                          00000500
                                    ++++++++++                                                                          00000600
                                    ++++++++++                                                                          00000700
                                    ++++++++++                                    DEFINE BIGTABSEM=[11                  00000800
                                    ++++++++++  15]#,BIGTABRIGHTPT=[35:13]#,BIGTABLEFTPT=[22:13]#,                      00000900
                                    ++++++++++  BIGTABCHARS=[19:3]#,BIGTABWORDS=[16:3]#,BIGTABSIGN=                     00001000
                                    ++++++++++  [16:1]#,BIGTABLOWBITS=[16:32]#,BIGTABHIGHBITS=[17:                      00001100
                                    ++++++++++  31]#,BIGTABCHAR1=[12:6]#,BIGTABCHAR2=[18:6]#,                          00001200
                                    ++++++++++  BIGTABCHAR3=[24:6]#,BIGTABCHAR4=[30:6]#,BIGTABCHAR5=                     00001300
                                    ++++++++++  [36:6]#,BIGTABCHAR6=[42:6]#,MSTACKSEM=[1:13]#,                         00001400
                                    ++++++++++  MSTACKTYPE=[32:4]#,MSTACKENTRY=[36:12]#}                               00001500
                                    ++++++++++        INTEGER PROCEDURE ISLSEARCH(O,E,P)}VALUE E,P}                     00001600
                                    ++++++++++  REAL O,E} INTEGER P} BEGIN LABEL X,D} FOR ISLI=P-1                      00001700
                                    ++++++++++  STEP -1 UNTIL 0 DO IF O=E THEN GO TO X} ISLSEARCH=-1                    00001800
                                    ++++++++++  }GO TO D} X:ISLSEARCH=ISLI} D:END OF ISLSEARCH}                        00001900
                                    ++++++++++  INTEGER ISLTEMP,ISLTEMP1,ISLTEMP2}                                     00002000
                                    ++++++++++                                                                          00002100
                                    ++++++++++                                                                          00002200
                                    ++++++++++                                                                          00002300
   INTEGER STORETYPE, CDLEVEL, PTABTAB, TEMP1, TEMP2, TEMP3 ,I }
                                    ++++++++++        INTEGER STORETYPE, CDLEVEL, PTABTAB, TEMP1, TEMP2, TEMP3 ,I }   00002400
   INTEGER MAXPTABTAB }
                                    ++++++++++        INTEGER MAXPTABTAB }                                            00002500
   DEFINE LAB=1#, BOOL=2#, INT=3#, LABELUSEDBUTNOTFOUND=6# }
                                    ++++++++++        DEFINE LAB=1#, BOOL=2#, INT=3#, LABELUSEDBUTNOTFOUND=6# }       00002600
   STABLE(IDTAB,1022,PTTOBIGTAB= 1:13,
                     BACKPT     =14:10,
                     PTTOABTAB  =24:8 ,
                     LEVEL      =32:4 ,
                     TYPE       =36:2 ,
                     USED       =38:1 ,
                     FOUND      =39:1 ,
                     LABFIELD   =36:4 ,
                     PTTOLKLIST=40:8 ,
                     REST       =38:10 }
          LKLIST,256,PTTOABTAB= 1:8 ,
                     LINK     = 9:8 }
          BLOCKSTACK,16,PTABTAB = 1:8 ,
                        PTLKLIST= 9:8 ) }
                                    ++++++++++  INTEGER PTIDTAB}REAL ARRAY IDTAB[0:1021]}DEFINE IDTABPTTOBIGTAB=[1:13]#,00002700
                                    ++++++++++  IDTABBACKPT=[14:10]#,IDTABPTTOABTAB=[24:8]#,IDTABLEVEL=[32:4]#,         00002800
                                    ++++++++++  IDTABTYPE=[36:2]#,IDTABUSED=[38:1]#,IDTABFOUND=[39:1]#,IDTABLABFIELD=[  00002900
                                    ++++++++++  36:4]#,IDTABPTTOLKLIST=[40:8]#,IDTABREST=[38:10]#}INTEGER PTLKLIST}     00003000
                                    ++++++++++  REAL ARRAY LKLIST[0:255]}DEFINE LKLISTPTTOABTAB=[1:8]#,LKLISTLINK=[9:8]#00003100
                                    ++++++++++  }INTEGER PTBLOCKSTACK}REAL ARRAY BLOCKSTACK[0:15]}DEFINE                 00003200
                                    ++++++++++  BLOCKSTACKPTABTAB=[1:8]#,BLOCKSTACKPTLKLIST=[9:8]#}                    00003300
```

```
$STACK(IFSTACK,32) ;
                                ++++++++++  INTEGER PTIFSTACK;REAL ARRAY IFSTACK[0:31];
$ADDCOL(BIGTAB,PTTOIDTAB=6:10 ,
           NUMBEROK =5:1  ;
        MSTACK,PTTOIDTAB=8:10 ,
           NUMBEROK =7:1 ) ;
                                ++++++++++  DEFINE BIGTABPTTOIDTAB=[6:10]#,BIGTABNUMBEROK=[5:1]#;DEFINE
                                ++++++++++  MSTACKPTTOIDTAB=[8:10]#,MSTACKNUMBEROK=[7:1]#;
$UPPERN(22) ;
FORMAT ERRORMESSAGE("***** SEMANTIC ERROR #",I1,
                                ++++++++++      FORMAT ERRORMESSAGE("***** SEMANTIC ERROR #",I1,
                  " , IDENTIFIER IS : ",8A6) ;
                                ++++++++++                          " , IDENTIFIER IS : ",8A6) ;
PROCEDURE ERROR(ERRORNUMBER,PTTOBIGTAB);
                                ++++++++++      PROCEDURE ERROR(ERRORNUMBER,PTTOBIGTAB);
  VALUE ERRORNUMBER,PTTOBIGTAB; INTEGER ERRORNUMBER,PTTOBIGTAB;
                                ++++++++++        VALUE ERRORNUMBER,PTTOBIGTAB; INTEGER ERRORNUMBER,PTTOBIGTAB;
  BEGIN
                                ++++++++++        BEGIN
    COMMENT SEMANTIC ERRORS IN DEMALGOL I :
                                ++++++++++           COMMENT SEMANTIC ERRORS IN DEMALGOL I :
      1 - ID HAS ALREADY BEEN DECLARED IN THIS BLOCK.
                                ++++++++++             1 - ID HAS ALREADY BEEN DECLARED IN THIS BLOCK.
      2 - A LABEL DECLARED AND USED IN A BLOCK DID NOT APPEAR.
                                ++++++++++             2 - A LABEL DECLARED AND USED IN A BLOCK DID NOT APPEAR.
      3 - LABEL WAS NOT DECLARED IN THE INNERMOST BLOCK IN WHICH
                                ++++++++++             3 - LABEL WAS NOT DECLARED IN THE INNERMOST BLOCK IN WHICH
          IT OCCURS.
                                ++++++++++                 IT OCCURS.
      4 - ID HAS NOT BEEN DECLARED AS A LABEL OR HAS NOT BEEN
                                ++++++++++             4 - ID HAS NOT BEEN DECLARED AS A LABEL OR HAS NOT BEEN
          DECLARED AT ALL.
                                ++++++++++                 DECLARED AT ALL.
      5 - ID HAS NOT BEEN DECLARED AS A BOOLEAN OR HAS NOT BEEN
                                ++++++++++             5 - ID HAS NOT BEEN DECLARED AS A BOOLEAN OR HAS NOT BEEN
          DECLARED AT ALL. ;
                                ++++++++++                 DECLARED AT ALL. ;
    TEMP3+$OPO(BIGTAB,PTTOBIGTAB,WORDS);
                                ++++++++++           TEMP3+
                                ++++++++++  BIGTAB[(PTTOBIGTAB).[9:30],(PTTOBIGTAB).[39:9]].BIGTABWORDS;
    WRITE(LINE,ERRORMESSAGE,ERRORNUMBER,FOR I+0 STEP 1 UNTIL TEMP3
                                ++++++++++           WRITE(LINE,ERRORMESSAGE,ERRORNUMBER,FOR I+0 STEP 1 UNTIL TEMP3
            DO $OPO(BIGTAB,PTTOBIGTAB+I+1,ALL));
                                ++++++++++                     DO
                                ++++++++++  BIGTAB[(ISLTEMP:=PTTOBIGTAB+I+1).[9:30],(ISLTEMP).[39:9]]);
    $INCR(NUMERRS)
                                ++++++++++  NUMERRS:=NUMERRS+1
  END OF PROCEDURE ERROR ;
                                ++++++++++        END OF PROCEDURE ERROR ;
                                ++++++++++
% ********** BEGINNING OF THE SEMANTIC ACTIONS **********
                                ++++++++++  % ********** BEGINNING OF THE SEMANTIC ACTIONS **********
                                ++++++++++
$ACTION( 0): CULEVEL+0; PTIDTAB+PTLKLIST+1; PTABTAB+2;
                                ++++++++++
                                ++++++++++  PROCEDURE EXEC(N);VALUE N;INTEGER N; BEGIN
                                ++++++++++   LABEL ISLEND, ACTION0, ACTION1, ACTION2, ACTION3, ACTION4, ACTION5,
                                ++++++++++   ACTION6, ACTION7, ACTION8, ACTION9, ACTION10, ACTION11, ACTION12,
                                ++++++++++   ACTION13, ACTION14, ACTION15, ACTION16, ACTION17, ACTION18, ACTION19,
```

```
                                    **********    ACTION9, ACTION10, ACTION11, ACTION12, ACTION13, ACTION14, ACTION15,    00006800
                                    **********    ACTION16, ACTION17, ACTION18, ACTION19, ACTION20, ACTION21, ACTION22;  00006900
                                    **********  GO TO ISLSWITCHLN+1]; ACTION0:                                            00007000
                                    **********  PTBIGTABI=0;PTIUTABI=0;PTLKLISTI=0;PTBLOCKSTACKI=0;PTIFSTACKI=0;          00007100
                                    **********  CULEVEL+0;PTIDTAB+PTLKLIST+1;PTABTAB+2;                                   00007200
            $CODE(0,BUOL,0,*GETSPACE,I,BUOL,1,*GETSPACE) ;
                                    **********    BEGIN  ISLOUTPUTCODE(0,TRUE); ISLOUTPUTCODE(BUOL,TRUE); ISLOUTPUTCODE(  00007300
                                    **********  0,TRUE); ISLOUTPUTCODE(66,FALSE); ISLOUTPUTCODE(I,TRUE); ISLOUTPUTCODE(  00007400
                                    **********  BUOL,TRUE); ISLOUTPUTCODE(1,TRUE); ISLOUTPUTCODE(66,FALSE) END ;          00007500
$ACTION( 1): $CODE(*ENDOFIT);
                                    **********    GO TO ISLEND; ACTION1: BEGIN  ISLOUTPUTCODE(67,FALSE) END ;              00007600
$ACTION( 2): FOR I+PTIDTAB-1 STEP -1 WHILE $OPD(IDTAB,I,LEVEL)=
                                    **********    GO TO ISLEND; ACTION2:FOR I+PTIUTAB-I STEP -1 WHILE IUTAB[I],            00007700
                                    **********  IUTABLEVEL=                                                               00007800
                                                        CULEVEL DO
                                    **********                                                          CULEVEL DO      00007900
          BEGIN
                                    **********                                  BEGIN                                   00008000
            $OPD(BIGTAB,$OPD(IDTAB,I,PTTOBIGTAB),PTTOIUTAB)+
                                    **********  BIGTAB[(ISLTEMPI=IUTAB[I],IDTABPTTOBIGTAB),(9:30),(ISLTEMP),(39:9)],    00008100
                                    **********  BIGTABPTTOIUTAB+                                                          00008200
                                                    $OPD(IDTAB,I,BACKPT) ;
                                    **********  IUTAB[I],IDTABBACKPT ;                                                    00008300
            IF $OPD(IDTAB,I,LABFIELD)=LABELUSEDBUTNOTFOUND
                                    **********                                    IF                                    00008400
                                    **********  IDTAB[I],IDTABLABFIELD=LABELUSEDBUTNOTFOUND                               00008500
              THEN ERROR(2,$OPD(IDTAB,I,PTTOBIGTAB)) ;
                                    **********                                      THEN ERROR(2,                       00008600
                                    **********  IDTAB[I],IDTABPTTOBIGTAB)];                                               00008700
            IF TEMP1+$OPD(IDTAB,I,TYPE)#LAB THEN
                                    **********                                    IF TEMP1+                             00008800
                                    **********  IDTAB[I],IDTABTYPE#LAB THEN                                               00008900
              $CODE($OPD(IUTAB,I,PTTOABTAB),
                    TEMP1,*+ORGETSPACE)];
                                    **********    BEGIN  ISLOUTPUTCODE(IDTAB[I],IUTABPTTOABTAB,TRUE); ISLOUTPUTCODE(      00009000
                                    **********  TEMP1,TRUE); ISLOUTPUTCODE(68,FALSE) END ;                                00009100
            PTIDTAB+I
                                    **********                                      PTIUTAB+I                           00009200
          END;
                                    **********                                  END;                                    00009300
        MAXPTABTAB+MAX(MAXPTABTAB,PTABTAB)];
                                    **********                              MAXPTABTAB+MAX(MAXPTABTAB,PTABTAB)];        00009400
        $POP(BLOCKSTACK,PTABTAB:PTABTAB,PTLKLIST:PTLKLIST)];
                                    **********  BEGIN PTBLOCKSTACK :=PTBLOCKSTACK-1;PTABTAB:=BLOCKSTACK[PTBLOCKSTACK]     00009500
                                    **********  ,BLOCKSTACKPTABTAB];PTLKLIST:=BLOCKSTACK[PTBLOCKSTACK],                   00009600
                                    **********  BLOCKSTACKPTLKLIST END ;                                                  00009700
        $DECR(CULEVEL)];
                                    **********  CULEVEL:=CULEVEL-1];                                                      00009800
$ACTION( 3): $INCR(CULEVEL)];
                                    **********    GO TO ISLEND; ACTION3:CULEVEL:=CULEVEL+1;                               00009900
        $PUSH(BLOCKSTACK,PTABTAB:PTABTAB,PTLKLIST:PTLKLIST)];
                                    **********  BEGIN BLOCKSTACK[PTBLOCKSTACK],BLOCKSTACKPTABTAB:=PTABTAB;BLOCKSTACK[     00010000
                                    **********  PTBLOCKSTACK],BLOCKSTACKPTLKLIST:=PTLKLIST;PTBLOCKSTACK :=                00010100
                                    **********  PTBLOCKSTACK+1 END ;                                                      00010200
$ACTION( 4): STORETYPE+INT ;
                                    **********    GO TO ISLEND; ACTION4:STORETYPE+INT ;                                   00010300
$ACTION( 5): STORETYPE+LAB ;
                                    **********    GO TO ISLEND; ACTION5:STORETYPE+LAB ;                                   00010400
$ACTION( 6): STORETYPE+BOOL ;
```

```
                                ++++++++++    GO TO ISLEND; ACTION6:STORETYPE=BOOL ;                                    00010500
$ACTION( 7): IF (IF TEMP1=$OPD(MSTACK,NOW,PTTOIDTAB)=0
                                ++++++++++    GO TO ISLEND; ACTION7:IF (IF TEMP1=MSTACK[PTMSTACK-1],                    00010600
                                ++++++++++  MSTACKPTTOIDTAB=0                                                           00010700
             THEN TRUE
                                ++++++++++                                THEN TRUE                                     00010800
             ELSE $OPD(IDTAB,TEMP1,LEVEL)<CULEVEL)
                                ++++++++++                                ELSE                                          00010900
                                ++++++++++  IDTAB[TEMP1],IDTABLEVEL<CULEVEL)                                            00011000
          THEN BEGIN
                                ++++++++++                           THEN BEGIN                                         00011100
               $EXEC(22);
                                ++++++++++    EXEC(22);                                                                00011200
               IF STORETYPE#LAB THEN
                                ++++++++++                                IF STORETYPE#LAB THEN                         00011300
                 $CODE(PTABTAB,STORETYPE,0,*GETSPACE);
                                ++++++++++    BEGIN  ISLOUTPUTCODE(PTABTAB,TRUE); ISLOUTPUTCODE(STORETYPE,TRUE);       00011400
                                ++++++++++    ISLOUTPUTCODE(0,TRUE); ISLOUTPUTCODE(66,FALSE) END ;                     00011500
               $INCR(PTABTAB)
                                ++++++++++  PTABTAB:=PTABTAB+1                                                          00011600
             END
                                ++++++++++                                END                                           00011700
          ELSE ERROR(1,$OPD(MSTACK,NOW,ENTRY));
                                ++++++++++                              ELSE ERROR(1,                                   00011800
                                ++++++++++  MSTACK[PTMSTACK-1],MSTACKENTRY);                                           00011900
$ACTION( 8): IF $OPD(IDTAB,TEMP1=$OPD(MSTACK,NOW,PTTOIDTAB),TYPE)=
                                ++++++++++    GO TO ISLEND; ACTION8:IF IDTAB[TEMP1=MSTACK[PTMSTACK-1],                  00012000
                                ++++++++++  MSTACKPTTOIDTAB],IDTABTYPE=                                                00012100
                                                    LAB
                                ++++++++++                                                                  LAB       00012200
          THEN IF $OPD(IDTAB,TEMP1,LEVEL)=CULEVEL
                                ++++++++++                              THEN IF                                         00012300
                                ++++++++++  IDTAB[TEMP1],IDTABLEVEL=CULEVEL                                            00012400
               THEN BEGIN
                                ++++++++++                                 THEN BEGIN                                   00012500
                    $CODE(TEMP3=$OPD(IDTAB,TEMP1,PTTOABTAB)
                          ,*STOREADD);
                                ++++++++++    BEGIN  ISLOUTPUTCODE(TEMP3=IDTAB[TEMP1],IDTABPTTOABTAB ,TRUE);           00012600
                                ++++++++++    ISLOUTPUTCODE(69,FALSE) END ;                                            00012700
                    TEMP2=$OPD(IDTAB,TEMP1,PTTOLKLIST);
                                ++++++++++                                          TEMP2=                              00012800
                                ++++++++++  IDTAB[TEMP1],IDTABPTTOLKLIST;                                               00012900
                    WHILE TEMP2#0 DO
                                ++++++++++                                          WHILE TEMP2#0 DO                    00013000
                    BEGIN
                                ++++++++++                                          BEGIN                               00013100
                      $CODE(TEMP3,
                            $OPD(LKLIST,TEMP2,PTTOABTAB),
                            *SETADDRUTRANSF );
                                ++++++++++    BEGIN  ISLOUTPUTCODE(TEMP3,TRUE); ISLOUTPUTCODE(                          00013200
                                ++++++++++  LKLIST[TEMP2],LKLISTPTTOABTAB,TRUE); ISLOUTPUTCODE(70,FALSE) END ;          00013300
                      TEMP2=$OPD(LKLIST,TEMP2,LINK)
                                ++++++++++                                            TEMP2=                            00013400
                                ++++++++++  LKLIST[TEMP2],LKLISTLINK                                                    00013500
                    END;
                                ++++++++++                                          END;                                00013600
                    $OPD(IDTAB,TEMP1,FOUND)=1
                                ++++++++++  IDTAB[TEMP1],IDTABFOUND=1                                                   00013700
                  END
                                ++++++++++                                       END                                    00013800
               ELSE ERROR(3,$OPD(MSTACK,NOW,ENTRY))
```

```
                                ++++++++++    ELSE ERROR(4,                                                    00014100
                                ++++++++++  MSTACK(PTMSTACK-1),MSTACKENTRY))                                   00014200
$ACTION( 9): IF $OPD(IDTAB,TEMP1+$OPD(MSTACK,NOW,PTTOIDTAB),TYPE)=
                                ++++++++++   GO TO ISLEND; ACTION9:IF IDTAB(TEMP1+MSTACK(PTMSTACK-1),.          00014300
                                ++++++++++  MSTACKPTTOIDTAB),IDTABTYPE=                                        00014400
                                                    LAB
                                ++++++++++                                                          LAB        00014500
             THEN BEGIN
                                ++++++++++                          THEN BEGIN                                 00014600
        IF $OPD(IDTAB,TEMP1,FOUND)=0 THEN
                                ++++++++++                                IF                                   00014700
                                ++++++++++  IDTAB(TEMP1),IDTABFOUND=0 THEN                                     00014800
          BEGIN
                                ++++++++++                                  BEGIN                              00014900
            $PUSH(LKLIST,PTTOABTAB:PTABTAB,
                  LINK:$OPD(IDTAB,TEMP1,PTTOLKLIST));
                                ++++++++++  BEGIN LKLIST(PTLKLIST),LKLISTPTTOABTAB:=PTABTAB;LKLIST(PTLKLIST),  00015000
                                ++++++++++  LKLISTLINK:=IDTAB(TEMP1),IDTABPTTOLKLIST;PTLKLIST :=PTLKLIST+1 END ; 00015100
            $ENTER(IDTAB,TEMP1,PTTOLKLIST:PTLKLIST=1,
                                    USED:1 );
                                ++++++++++  BEGIN IDTAB(TEMP1),IDTABPTTOLKLIST:=PTLKLIST-1;IDTAB(TEMP1),IDTABUSED  00015200
                                ++++++++++  :=1 ;END ;                                                         00015300
            $CODE(PTABTAB,*STOREADDR); $INCR(PTABTAB)
                                ++++++++++   BEGIN  ISLOUTPUTCODE(PTABTAB,TRUE); ISLOUTPUTCODE(71,FALSE) END ; 00015400
                                ++++++++++  PTABTAB:=PTABTAB+1                                                 00015500
          END;
                                ++++++++++                                  END;                               00015600
        $CODE($OPD(IDTAB,TEMP1,PTTOABTAB),*UTRANSF)
                                ++++++++++   BEGIN  ISLOUTPUTCODE(IDTAB(TEMP1),IDTABPTTOABTAB,TRUE); ISLOUTPUTCODE( 00015700
                                ++++++++++  72,FALSE) END                                                      00015800
      END
                                ++++++++++                              END                                    00015900
    ELSE ERROR(4,$OPD(MSTACK,NOW,ENTRY)));
                                ++++++++++                            ELSE ERROR(4,                            00016000
                                ++++++++++  MSTACK(PTMSTACK-1),MSTACKENTRY))                                   00016100
$ACTION(10): SEMANTICTEST+$OPD(IDTAB,$OPD(MSTACK,NOW,PTTOIDTAB),
                                ++++++++++   GO TO ISLEND; ACTION10:SEMANTICTEST+                              00016200
                        TYPE) = INT ;
                                ++++++++++  IDTAB(MSTACK(PTMSTACK-1),MSTACKPTTOIDTAB),IDTABTYPE =INT ;        00016300
$ACTION(11): $CODE($OPD(IDTAB,$OPD(MSTACK,PTMSTACK-3,PTTOIDTAB),
                                ++++++++++   GO TO ISLEND; ACTION11:                                           00016400
          PTTOABTAB),INT,*STORE);
                                ++++++++++   BEGIN  ISLOUTPUTCODE(                                           00016500
                                ++++++++++  IDTAB(MSTACK(PTMSTACK-3),MSTACKPTTOIDTAB),IDTABPTTOABTAB,TRUE);  00016600
                                ++++++++++   ISLOUTPUTCODE(INT,TRUE); ISLOUTPUTCODE(73,FALSE) END ;          00016700
$ACTION(12): IF $OPD(IDTAB,TEMP1+$OPD(MSTACK,PTMSTACK-3,PTTOIDTAB),
                                ++++++++++   GO TO ISLEND; ACTION12:IF                                         00016800
                 TYPE)=BOOL
                                ++++++++++  IDTAB(TEMP1+MSTACK(PTMSTACK-3),MSTACKPTTOIDTAB),IDTABTYPE=BOOL    00016900
        THEN $CODE($OPD(IDTAB,TEMP1,PTTOABTAB),BOOL,*STORE)
                                ++++++++++                            THEN                                     00017000
                                ++++++++++   BEGIN  ISLOUTPUTCODE(IDTAB(TEMP1),IDTABPTTOABTAB,TRUE); ISLOUTPUTCODE( 00017100
                                ++++++++++  BOOL,TRUE); ISLOUTPUTCODE(73,FALSE) END                           00017200
        ELSE ERROR(5,$OPD(MSTACK,PTMSTACK-3,ENTRY)));
                                ++++++++++                            ELSE ERROR(5,                            00017300
                                ++++++++++  MSTACK(PTMSTACK-3),MSTACKENTRY))                                   00017400
$ACTION(13): $PUSH(IFSTACK:PTABTAB));
                                ++++++++++   GO TO ISLEND; ACTION13:BEGIN IFSTACK(PTIFSTACK):=PTABTAB;PTIFSTACK := 00017500
```

```
                                    ++++++++++  =PTIFSTACK+1 END }                                                    00017600
          $CODE(PTABTAB,*STOREADDR,0,*CTRANSF)}
                                    ++++++++++    BEGIN  ISLOUTPUTCODE(PTABTAB,TRUE)} ISLOUTPUTCODE(71,FALSE)}      00017700
                                    ++++++++++    ISLOUTPUTCODE(0,TRUE)} ISLOUTPUTCODE(74,FALSE) END }              00017800
          $INCR(PTABTAB)}
                                    ++++++++++  PTABTAB:=PTABTAB+1}                                                  00017900
$ACTION(14): $POP(IFSTACK:TEMP1)} TEMP2:=PTABTAB+1}
                                    ++++++++++    GO TO ISLEND} ACTION14:BEGIN PTIFSTACK :=PTIFSTACK-1}TEMP1:=IFSTACK[  00018000
                                    ++++++++++  PTIFSTACK]END }TEMP2:=PTABTAB+1}                                     00018100
          $CODE(PTABTAB,*STOREADDR,0,*UTRANSF,TEMP2,*STOREADDR,
                TEMP2,TEMP1,*SETADDRCTRANSF)}
                                    ++++++++++    BEGIN  ISLOUTPUTCODE(PTABTAB,TRUE)} ISLOUTPUTCODE(71,FALSE)}      00018200
                                    ++++++++++    ISLOUTPUTCODE(0,TRUE)} ISLOUTPUTCODE(72,FALSE)} ISLOUTPUTCODE(TEMP2  00018300
                                    ++++++++++  ,TRUE)} ISLOUTPUTCODE(71,FALSE)} ISLOUTPUTCODE(TEMP2,TRUE)}          00018400
                                    ++++++++++    ISLOUTPUTCODE(TEMP1,TRUE)} ISLOUTPUTCODE(75,FALSE) END }          00018500
          $PUSH(IFSTACK:PTABTAB)} $INCR(PTABTAB)}
                                    ++++++++++  BEGIN IFSTACK[PTIFSTACK]:=PTABTAB}PTIFSTACK :=PTIFSTACK+1 END }PTABTAB:=00018600
                                    ++++++++++  PTABTAB+1}                                                           00018700
$ACTION(15): $CODE(PTABTAB,*STOREADDR,PTABTAB,$OPD(IFSTACK,NOW),
                                    ++++++++++    GO TO ISLEND} ACTION15:                                           00018800
                *SETADDRUTRANSF)}
                                    ++++++++++    BEGIN  ISLOUTPUTCODE(PTABTAB,TRUE)} ISLOUTPUTCODE(71,FALSE)}      00018900
                                    ++++++++++    ISLOUTPUTCODE(PTABTAB,TRUE)} ISLOUTPUTCODE(IFSTACK[PTIFSTACK-1],TRUE)} 00019000
                                    ++++++++++    ISLOUTPUTCODE(70,FALSE) END }                                     00019100
          $POP(IFSTACK)}
                                    ++++++++++  BEGIN PTIFSTACK :=PTIFSTACK-1 END }                                  00019200
$ACTION(16): $CODE(#$OPD(MSTACK,PTMSTACK-2,ENTRY))}
                                    ++++++++++    GO TO ISLEND} ACTION16: BEGIN  ISLOUTPUTCODE(                     00019300
                                    ++++++++++  MSTACK[PTMSTACK-2],MSTACKENTRY,FALSE) END }                         00019400
$ACTION(17): $CODE($OPD(IOTAB,$OPD(MSTACK,NOW,PTTOIOTAB),
                                    ++++++++++    GO TO ISLEND} ACTION17:                                           00019500
                                          PTTOABTAB))}
                                    ++++++++++    BEGIN  ISLOUTPUTCODE(                                             00019600
                                    ++++++++++  IOTAB[MSTACK[PTMSTACK-1],MSTACKPTTUIOTAB],IOTABPTTUABTAB,TRUE) END }  00019700
$ACTION(18): IF TEMP1+$OPD(MSTACK,NOW,NUMBEROK)=0
                                    ++++++++++    GO TO ISLEND} ACTION18:IF TEMP1+MSTACK[PTMSTACK-1],MSTACKNUMBEROK=0  00019800
    THEN BEGIN
                                    ++++++++++                          THEN BEGIN                                   00019900
            $EXEC(22)}
                                    ++++++++++    EXEC(22)}                                                         00020000
            $OPD(BIGTAB,TEMP2,NUMBEROK)+1}
                                    ++++++++++  BIGTAB[(TEMP2),[9:30],(TEMP2),[39:9]],BIGTABNUMBEROK+1}              00020100
            $CODE(PTABTAB,INT,
                  $OPD(BIGTAB,TEMP2+1,ALL),
                  *GETSPACE,PTABTAB)}
                                    ++++++++++    BEGIN  ISLOUTPUTCODE(PTABTAB,TRUE)} ISLOUTPUTCODE(INT,TRUE)}      00020200
                                    ++++++++++    ISLOUTPUTCODE(BIGTAB[(ISLTEMP:=TEMP2+1),[9:30],(ISLTEMP),[39:9]],TRUE)}00020300
                                    ++++++++++    ISLOUTPUTCODE(66,FALSE)} ISLOUTPUTCODE(PTABTAB,TRUE) END }         00020400
            $INCR(PTABTAB)
                                    ++++++++++  PTABTAB:=PTABTAB+1                                                   00020500
          END
                                    ++++++++++                                END                                    00020600
    ELSE $CODE($OPD(IOTAB,$OPD(MSTACK,NOW,PTTOIOTAB),
                                    ++++++++++                              ELSE                                   00020700
                    PTTUABTAB))}
                                    ++++++++++    BEGIN  ISLOUTPUTCODE(                                             00020800
                                    ++++++++++  IOTAB[MSTACK[PTMSTACK-1],MSTACKPTTUIOTAB],IOTABPTTUABTAB,TRUE) END }  00020900
$ACTION(19): IF $OPD(IOTAB,TEMP1+$OPD(MSTACK,NOW,PTTOIOTAB),TYPE)=
                                    ++++++++++    GO TO ISLEND} ACTION19:IF IOTAB[TEMP1+MSTACK[PTMSTACK-1],.          00021000
                                    ++++++++++  MSTACKPTTOIOTAB],IOTABTYPE=                                          00021100
                                                  BOOL
```

```
                                    ++++++++++                    THEN                                                    00021300
                                    ++++++++++    BEGIN  ISLOUTPUTCODE(IDTAB[TEMP1],IDTABPTTOABTAB,TRUE) END         00021400
                ELSE ERROR(5,$OPD(MSTACK,NOW,ENTRY))$
                                    ++++++++++                    ELSE ERROR(5,                                           00021500
                                    ++++++++++  MSTACK[PTMSTACK-1],MSTACKENTRY)$                                          00021600
    $ACTION(20)$ $CODE(1)$
                                    ++++++++++   GO TO ISLEND$ ACTION20$ BEGIN  ISLOUTPUTCODE(1,TRUE) END $             00021700
    $ACTION(21)$ $CODE(0)$
                                    ++++++++++   GO TO ISLEND$ ACTION21$ BEGIN  ISLOUTPUTCODE(0,TRUE) END $             00021800
    $ACTION(22)$ $OPD(BIGTAB,TEMP2+$OPD(MSTACK,NOW,ENTRY),PTTOIDTAB)+
                                    ++++++++++   GO TO ISLEND$ ACTION22$BIGTAB[(ISLTEMP$=TEMP2+MSTACK[PTMSTACK-1],      00021900
                                    ++++++++++  MSTACKENTRY),[9:30],(ISLTEMP),[39:9]],BIGTABP[TOIDTAB+                  00022000
                                                      PTIDTAB$
                                    ++++++++++                                                          PTIDTAB$      00022100
                $PUSH(IDTAB,PTTOBIGTAB:TEMP2,
                                BACKPT    :TEMP1,
                                PTTOABTAB :PTABTAB,
                                LEVEL     :CULEVEL,
                                TYPE      :STORETYPE,
                                REST      :0 )
                                    ++++++++++  BEGIN IDTAB[PTIDTAB].IDTABPTTOBIGTAB:=TEMP2$IDTAB[PTIDTAB],             00022200
                                    ++++++++++  IDTABBACKPT:=TEMP1$IDTAB[PTIDTAB],IDTABPTTOABTAB:=PTABTAB$IDTAB[       00022300
                                    ++++++++++  PTIDTAB],IDTABLEVEL:=CULEVEL$IDTAB[PTIDTAB],IDTABTYPE:=STORETYPE$      00022400
                                    ++++++++++  IDTAB[PTIDTAB],IDTABREST:=0 $PTIDTAB :=PTIDTAB+1 END                   00022500
$END
                                    ++++++++++   $ GO TO ISLEND$ISLEND: END OF PROCEDURE EXEC  $                        00022600
                                    ++++++++++  LANGNAME:="DEMALGI"$ NAMEA:="     D"$                               00022700
                                    ++++++++++  NAMEB:="EMALGI"$ VERSION:=  1$USINGILCODE:=TRUE                        00022800
                                    ++++++++++  $ PARSE$END OF PARSER BLOCK$     END.                                00022900
                                    ++++++++++                                                                          00023000
                                    ++++++++++                                                                          00023100
                                    ++++++++++                                                                          00023200
                                    ++++++++++                                                                          00023300
                                    ++++++++++                                                                          00023400
                                    ++++++++++                                                                          00023500
                                    ++++++++++                                                                          00023600
                                    ++++++++++                                                                          00023700
                                    ++++++++++                                                                          00023800
                                    ++++++++++                                                                          00023900
                                    ++++++++++                                                                          00024000


        CONGRATULATIONS,,,,TRANSLATION COMPLETED SUCCESFULLY


        THE TRANSLATED FILE IS ON DISK UNDER THE NAME: DEMALGI/ALGSEM
        COMPILATION OF THE COMPILER WAS INITIATED UNDER THE NAME: ALGOL/DEMALGI
        IF THIS COMPILATION IS SUCCESFUL , THE COMPILER WILL BE PUT ON DISK UNDER THE NAME: DEMALGI/COMPILR


        TOTAL TIME :         51 SECONDS
        PROCESSOR TIME :         21 SECONDS
        I/O TIME :        29 SECONDS
```

LIST OF REFERENCES

[1] Burroughs B-5500 Information Processing Systems, "Extended Algol Reference Manual", publication No. 1028024, Burroughs Corporation (1968).

[2] Beals, A. J., "The Generation of a Deterministic Parsing Algorithm", Report No. 304, Department of Computer Science, University of Illinois at Urbana-Champaign (January, 1969).

[3] Mercer, R., "TWINKLE--A Syntax Language for a Translator Writing System", M.S. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (December,1969).

[4] Machado, N. C., and Northcote, R. S., "The Structure of a Multi-pass Translator Writing System", to be published.

[5] Naur, P. (Ed.), "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM, 6 (January, 1963), pp. 1-23.

[6] Feldman, J., and Gries, D., "Translator Writing Systems", Comm. ACM, 11 (February, 1968), pp. 77-133.

[7] Floyd, R. W., "Assigning Meaning to Programs", Proc. Symp. Appl. Math., 19, AMS (1967), pp. 19-31.

[8] Landin, P. J., "The Mechanical Evaluation of Expressions", Comp. Journal, 6, (January, 1964), pp. 308-320.

[9] Rutledge, J. J., "On Ianov's Program Schemata", Journal of the ACM, 11 (January, 1964), pp. 1-9.

[10] Feldman, J. A., "A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler", Comm. ACM, 9 (Jan., 1966), pp. 3-9.

[11] Northcote, R. S., "A Formal Semantics Language (FSL) to Facilitate the Computer Translation of Programming Languages", File No. 621, Department of Computer Science, University of Illinois at Urbana-Champaign (1964).

[12] Iturriaga, R., et al., "Techniques and Advantages of Using the Formal Compiler Writing System FSL to Implement a Formula ALGOL Compiler", Proc. of the Spring Joint Computer Conference (1966), pp. 241-252.

[13] Iturriaga, R., et al., "The Implementation of Formula ALGOL in FSL", Carnegie-Mellon University, Pittsburgh, Pennsylvania (1966).

[14] Northcote, R. S., "The Structure and Use of a Compiler-Compiler System", Proc. of the Third Australian Computer Conference (May, 1966), pp. 339-344.

[15] Trout, R. G., "A Compiler-Compiler System", Proc. ACM 22nd National Conference (1967), pp. 317-322.

[16] Trout, R. G., "A BNF like Language for the Description of Syntax Directed Compilers", Report No. 300, Department of Computer Science, University of Illinois at Urbana-Champaign (Jan., 1969).

[17] Brooker, R. A., Morris, D., and Rohl, J. S., "Experience with the Compiler-Compiler", Comput. Journal, 9 (1967), pp. 345-349.

CLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

ORIGINATING ACTIVITY (Corporate author)

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

2a. REPORT SECURITY CLASSIFICATION

UNCLASSIFIED

2b. GROUP

REPORT TITLE

ISL-A SEMANTICS LANGUAGE FOR A TRANSLATOR WRITING SYSTEM

DESCRIPTIVE NOTES (Type of report and inclusive dates)

Research Report

AUTHOR(S) (First name, middle initial, last name)

Nelson C. Machado

REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
December 11, 1969	163	17

CONTRACT OR GRANT NO.

AF 30(602)-4144

PROJECT NO.

46-26-15-305

9a. ORIGINATOR'S REPORT NUMBER(S)

DCS Report No. 367

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DCS.

SUPPLEMENTARY NOTES

None

12. SPONSORING MILITARY ACTIVITY

Rome Air Development Center
Griffiss Air Force Base
Rome, New York 13440

ABSTRACT

It is well known that a Translator Writing System (TWS) can, in general, be divided into two parts: a syntax preprocessor and a semantics preprocessor, yielding compilers composed of a syntactic recognizer and a package of semantic routines. This paper describes the semantic preprocessor of a TWS. Semantics is specified as a program written in a special purpose language called ISL which is basically an extension of ALGØL. The extensions include some constructs often needed in the description of semantics of a language, such as stack and table declaration and manipulation, and control words to automate to a great extent the use of the TWS. The structure and use of ISL are described and an example of the use of ISL to specify the semantics of a small subset of ALGØL is presented and discussed.

D FORM 1 NOV 65 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
translator writing system						
semantics						
compiler						
semantic meta-language						
semantic description						
compiler compiler						